

嵌入式系统的 C 程序设计



Powered by xiaoguo's publishing studio
QQ:8204136

目 录



译者序

第1章 简介.....1

1.1 本书的作用1

1.2 嵌入式系统中使用C语言的好处.....1

1.3 本书概览2

1.4 修改和补充信息2

第2章 问题规范3

2.1 产品需求3

2.2 硬件管理3

2.3 软件计划4

2.3.1 软件体系结构5

2.3.2 伪代码6

2.3.3 流程图6

2.3.4 状态图6

2.3.5 资源管理8

2.4 测试规划8

第3章 微控制器10

3.1 中央处理单元11

3.1.1 指令集11

3.1.2 栈12

3.2 内存寻址和类型12

3.2.1 RAM和ROM13

3.2.2 ROM和程序设计13

3.2.3 冯·诺依曼与哈佛体系结构14

3.3 定时器14

3.3.1 监视定时器15

3.3.2 实例15

3.4 中断电路16

3.4.1 向量和非向量仲裁16

3.4.2 中断期间保存状态17

3.4.3 执行中断处理程序18

3.4.4 多个中断19

3.4.5 RESET19

3.5 I/O端口19

3.6 串行外设总线21

3.7 微控制器的开发工具21

第4章 设计过程23

4.1 产品功能23

4.2 硬件设计23

4.3 软件设计24

4.3.1 软件体系结构24

4.3.2 流程图24

4.4 资源管理25

4.4.1 暂存缓冲器26

4.4.2 中断计划26

4.5 测试选择27

4.5.1 调试规划27

4.5.2 代码检查27

4.5.3 模拟器环境28

4.5.4 仿真器环境28

4.5.5 在测试套件里的目标系统28

第5章 嵌入式系统中使用C29

5.1 内联汇编语言29

5.2 设备知识30

5.2.1 #pragma has30

5.2.2 #pragma port32

5.2.3 字节次序32

5.3 机械知识33

5.4 函数库34

5.5 初看嵌入式C程序34

第6章 数据类型和变量36

6.1 标识符声明37

6.2 函数数据类型37

6.3 字符数据类型38

6.4 整数数据类型	38	7.6 操作符和表达式	56
6.5 位数据类型	39	7.6.1 标准数学操作符	56
6.6 实数	39	7.6.2 位逻辑操作符	56
6.7 复杂数据类型	40	7.6.3 移位操作符	58
6.7.1 指针	40	第8章 函数库	60
6.7.2 数组	40	8.1 创建函数库	60
6.7.3 枚举类型	41	8.2 编写函数库	62
6.7.4 结构	42	8.3 函数库与链接	64
6.7.5 联合	43	第9章 优化和测试嵌入式C语言程序	67
6.8 typedef	44	9.1 优化	67
6.9 数据类型修饰符	45	9.2 手工优化	68
6.9.1 数值常量修饰符: const和 volatile	45	9.3 调试嵌入式C语言程序	70
6.9.2 允许值修饰符: signed和 unsigned	46	9.3.1 寄存器类型的修饰符	70
6.9.3 大小修饰符: short和long	46	9.3.2 局部内存	70
6.9.4 指针范围修饰符: near和far	46	9.3.3 指针	70
6.10 存储类修饰符	47	9.4 混合C语言和汇编语言	71
6.10.1 外部链接	47	9.4.1 调用规范	71
6.10.2 内部链接	47	9.4.2 从汇编代码中访问C变量	71
6.10.3 无链接	47	9.5 试验硬件	71
6.10.4 extern修饰符	48	9.6 通过检查调试	71
6.10.5 static修饰符	48	9.7 假载荷	73
6.10.6 register修饰符	49	9.8 仿真器和模拟器的运用	73
6.10.7 auto修饰符	49	9.8.1 模拟器	73
第7章 C语言的语句、结构及操作	51	9.8.2 仿真器	73
7.1 块中的联合语句	51	9.9 嵌入式软件的封装	74
7.2 函数	51	第10章 样例工程	75
7.3 控制结构	52	10.1 硬件的练习程序	75
7.3.1 main()函数	52	10.1.1 显示“Hello World!”	75
7.3.2 初始化函数	52	10.1.2 键盘测试	76
7.3.3 控制语句	52	10.1.3 LCD测试	77
7.4 选择结构	52	10.2 与端口通信	78
7.5 循环结构	54	10.3 A/D转换器原理	78
7.5.1 控制表达式	54	附录A 嵌入式C语言函数库	81
7.5.2 break和continue	54	附录B ASCII码表	107
		附录C 术语表	108
		光盘内容	III

第1章 简介

1.1 本书的作用

本书提供了使用C编程语言进行微控制器程序设计的一个完整的中级讨论，覆盖了设计嵌入式环境所需对C的改编，以及一个成功开发工程的通用组成部分的全部内容。

C是编写基于32位内核的较大微控制器（MCU）所选择的语言。这些微控制器通常由它们的通用微控制器衍生而来，并且同通用微控制器一样，既复杂又功能丰富。因此，对于这些MCU，C（和C++）编译器是必需的，也是很容易得到的。

相反，选择采用8位控制器的设计者经常求助于汇编语言的手工编码。虽然用于精确控制的手工汇编程序设计从来都不会过时，但也不会推动降低成本。因此，即使在8位MCU的有限资源里，编译高级C语言仍然有许多优势。

- 对如16位或更长的数据类型的算法之类的重复编码任务能自动生成代码。
- 硬件特殊性的直观处理。对一个串行闪存设备的读或写能用C语言表达为一个简单的赋值语句，尽管存储操作需要一些编码。
- 平台独立性。C带给桌面计算的跨平台能力对目前市场上的8位微控制器领域也是同样适用的。

本书将展示怎样用C语言编写8位嵌入式MCU程序。我们希望您不仅熟悉C，同时还具备有关微控制器程序设计更深层次的知识。

本书的主要样例工程是计算机控制的自动调温器。从一个最初规范开始，我们用与其他任何消费品或控制产品相同的方式逐步求精和增加设备。软件开发是我们关注的焦点，我们将做出任何设计者将要做出的选择和权衡。

1.2 嵌入式系统中使用C语言的好处

在嵌入式系统中使用C语言的好处如下：

不会困于细节。8位微控制器不仅仅是外形小；微控制器只包括执行其限定的任务所必需的逻辑，而程序员的付出是“轻松”的。通过一个C编译器与这些有限的资源协调工作，有助于抽象体系结构和避免坠入操作代码序列和硅片故障的云里雾里。

学习可移植性的原理。嵌入式应用程序是成本敏感的。为减少每个单元的成本，改变部件（或者甚至体系结构）可能是最大的激励。然而，修改汇编代码使得针对一种微处理器编写的程序能够在不同的处理器上运行，这样降低成本可能打消做出这种改变的任何念头。

通过传统编程降低成本。本书强调C语言代码能够概括微控制器特征。与特定硬件实施相关的细节能够被放置在不同的库函数和头文件中。使用C库函数和头文件可保证应用程序源代码能

够针对不同微控制器目标重新编译。

花费在算法设计上的时间更多而在实现上的时间更少。C是一种高级语言，使用它能够快速而轻松地编写应用程序。C的表达范围是简明而强大的。因此，用C语言编写的每行代码能够代替多行汇编语言代码。调试和维护C语言代码将比汇编代码容易得多。

1.3 本书概览

确定软件开发目标是第1步，将在第2章中论述。它包括有关对高效软件开发至关重要的预设计文档规则的嵌入式注解。

第3章为以前没有涉及过8位微控制器的读者提供一个由浅入深的介绍。

有了一个好的计划和关于中央控制器的深入知识后，设计过程（在第4章论述）把以前的评估都最后确定下来。与实现自动调温器有关的处理器细节也在第4章介绍。

第5章详细描述了硬件的C语言表达。它汇集了编写程序源码所必需的所有设置。

第6章提供对嵌入式数据的深刻剖析。变量存储修饰符near和far在运行微软视窗的Intel PC上和运行您的代码的嵌入式处理器上将代表不同的事物。

第7章讲述C语句，提供关于嵌入式的函数、语句和操作符的信息。

第8章介绍函数库。即使在只有很少ROM和有极其特定工作要做的环境里，预先编写的函数库也会带来很大的帮助。

第9章提供关于代码优化方面的深入知识，并帮助您彻底测试您创造的产品。

第10章总结了样例工程的更多信息。尽管某些信息已经在本章前面出现过，但它包括以前没有讨论的内容。

1.4 修改和补充信息

如果需要查找关于自动调温器工程的更多信息，请通过web咨询我们的补充信息：
http://www.bytecraft.com/embedded_C/。

第2章 问题规范

问题规范是设备和软件将要解决的问题的最初文档，它不应当包括任何具体设计难题或者产品解决方案，其主要的目的是详细解释程序将要做什么。

当然，正如在这个星球上有许多工作场所一样，处理工程计划的方法也有许多种。即使在那些最标准化了的阶段，也体现出不同的风格或者不同的次序。下面各节之所以被包含进来是因为它们增添了嵌入式领域的信息，有的则特别地适合样例工程。

2.1 产品需求

通常，这个文档是从用户的视角来写的，由一系列的用户需求组成。就针对一单任务设计的嵌入式系统的情况而言，应相当明确和肯定产品预期的功能范围。

有关硬件的总体决策是问题规范的组成部分，尤其在硬件将要受到充分控制的嵌入式工程中。

结果

- 程序将测量和显示当前的温度。
- 程序将按12或24小时制计数实际时间，并在一个数字显示屏上显示小时和分钟。
- 程序将接受时间设置并设置时钟。
- 程序将为三个日常使用周期接受和存储时间设置。
- 程序将控制制冷和制热之间的切换。注意，某些HVAC（heating ventilating and air conditioning，供热通风及空调）专家可能会想到偶尔在同一时间里既操作制冷又操作制热的需要，但这里的需求更接近于传统的自动调温器操作。
- 程序将比较当前温度与当前时间周期的设置，进而按需要打开或关闭外部的制冷或制热单元。
- 程序将抑制在一个短周期时间里两次改变外部单元的状态，这是为了使HVAC设备运行良好。
- 程序将在任何时间接受人工干预，进而立即关闭制热或制冷单元。

2.2 硬件管理

除了样例工程，本书并不直接涉及硬件。尽管如此，目标平台影响到产品的每个方面。它决定由编译器产生代码的容易程度，并且决定某些整体软件的设计决策。

如果软件开发者非常幸运，已经参与了硬件开发过程，那么影响设计的机会就太重要了且不要错过。请求的意向清单项目如下所示：

内置调试接口 现场可编程（field-programmability）的另一种方法也能满足需要。当一个设备必须现场安装、定制或修补时，选用Flash-ROM芯片将比选用EEPROM或ROM设备更明智。

ROM代码保护 嵌入式处理器往往提供ROM代码保护，以防止偶然的检查。一个配置位禁止通过程序设计接口读ROM。尽管存在几个破坏这项保护的方法，但只有某个致命的对手才能够成功地读到您的程序。

合理的外设接口 电子线路设计时因贪图方便而影响I/O组织时，可能会迅速地降低软件性能。首选处理器有独立改变端口比特的位操作指令吗？多路复用接口将需要很多数据方向切换吗？

通过采用通用I/O端口线路和驱动器软件，某些外设能够被（软）复制。这节省了钱但增加了程序设计任务的复杂性。为模仿专用外设电路的逻辑信号，软件必须快速而重复地写比特序列到端口输出线路，这被典型地描述为比特风暴（bit-banging）。

标准函数库也许不会产生特别优化的硬件方案，但能够用减少的软件成本来补偿增加的硬件成本。

硬件设计的中心决策是处理器选择。处理器的选定是一个协商的决定，权衡的因素有：预期应用所必需的资源、芯片供应的价格和可用性以及可得到的开发工具等。针对微控制器的深入探讨请参见下一章。内存估计是问题规范的组成部分，因此RAM和ROM大小的估计在2.3.5节的资源管理中讨论。

结果

本书不涉及硬件管理，但本书包括了实现信息设置的硬件的某些样例产品规范信息。

表2-1 初始硬件规范

管理因素	评 估
操作环境	<ul style="list-style-type: none"> • 局部环境 • 中等功耗、中等噪声电子线路
接口	<ul style="list-style-type: none"> • 偶尔断电 • 一个针对切换HVAC的多位端口：也许只有3个针脚是必需的 • 一个针对显示屏的多位I/O接口 • 一个针对键盘的多位I/O接口 • 一个针对温度传感的A/D设备 • 实时时钟源：1秒粒度
内存大小	(参见下文)
特殊功能	<ul style="list-style-type: none"> • 时钟/计数器或者实时时钟 • NVRAM的使用依赖于处理器是否和怎样睡眠 • 监视定时器也许是有用的
开发工具	<ul style="list-style-type: none"> • C编译器 • 模拟器或仿真器 • 开发板

2.3 软件计划

软件计划应该对程序设计语言的选择做某些说明。对于嵌入式系统，通常的开发语言有三种选择：机器语言、C语言或者像BASIC那样的某种高级语言。三种选择当中，C平衡了两个对抗的需求。

- 比起像BASIC语言那样的解释系统，C有接近手工编码的机器语言的性能。如果一个BASIC系统通过暴露指针或通过预编译源代码变得更加复杂，则在测试时的困难将与C不相上下。
- C提供了机器语言没有提供的设备独立性。如果用汇编语言手工编码一个程序，则会遇到随微控制器的改变而废弃全部代码的风险。用C编程在改变处理器时只需要付出很小的努力，在软件模块里修改一个头文件即可。

软件计划中的第1步是选择解决在问题规范中所描述问题的算法。各种不同的算法应当被考虑到，并且依照它们的代码大小、速度、难度以及维护成本作出比较。

一旦一个基本的算法被选定，整个问题应该被分解为多个更小的问题。家庭自动调温器工程很自然地细分为对应于下面每个设备的模块以及每个设备的每一种函数：

- HVAC接口
- 键盘
- LCD
- 温度传感器

从模块开始工作，可以写传统的伪代码。这有助于产生将要在代码中实现的标识符和逻辑段落。

从流程图着手把自然语言伪代码翻译为实际代码。在流程图里可以开始重点考虑函数将接受和提供的数据。最重要的是可以开始计划函数库的使用了。即使这里没有预写的外设或数据转换库存在，我们也能够以库的形式编写原始代码并在将来更加易于重用。

把不同的状态引入到计划中是可能的。状态框图映射了这种状态转换，它可以作为流程图的一种补充工具。

从伪代码开始，能够建立一个变量清单并对RAM和ROM的需要作出评估。内存资源的限制将对某些程序员是一个打击。用现代桌面环境工作的程序员在巨大的内存空间里很舒适，大量的RAM可用来创建可能永远也不会被初始化或用到的大数据结构或数组。

相反，微控制器只能在特定类目标应用所需要的那么多RAM和ROM里运转。制造商力求提供一系列类似的部件，每个变种的贡献只是增加少量的单片资源。

结果

2.3.1 软件体系结构

自动调温器设备的程序设计语言是C。主要体系结构的难以决断之处涉及采用中断还是轮询（或查询）。决断当中部分将随芯片选择而自动决定：一些处理器变种根本就不包括中断。其他选择包括对中断驱动的键盘的显式支持，或者在超时产生中断的定时器。

一个基于中断方案的重要方面是在中断和主程序代码之间的通信协议。因为中断和主程序是尽可能独立的（一个中断可能在任何主程序指令期间出现），所以竞争条件是出现的一个结果。

我们已经选定几个替代算法中最简单的那个：一个时钟/计数器中断将计算时间，请求显示屏修改和设置目标温度。主程序将循环查询键盘，采样环境温度，修改显示屏和切换HVAC机械。这一切只需要一个精确的计时中断，它必须24小时不间断工作。

2.3.2 伪代码

伪代码用自然语言表达程序的必要步骤。它在嵌入式程序设计里显得特别有用，因为执行的每个方面能够一起规划，而没有必要考虑操作系统的古怪特性。

在下面的示例里，假定时间是用计数器和软件跟踪的。

1. 初始化

- 1) 设置时钟计数器为0。
- 2) 设置时间和温度目标变量为默认值。
- 3) 启用时间中断。

2. 时钟/计数器每秒触发中断一次。

- 1) 增加时钟计数器。
- 2) 请求显示屏修改。
- 3) 循环遍历预置周期。如果时钟正好在或过了指明的周期时间，设置目标温度为那个周期的温度。

3. 主循环

- 1) 采样环境温度。
 - (a) 如果环境温度在目标温度范围之外，打开制冷或制热开关。
 - (b) 如果环境温度在目标温度范围之内，关闭制冷或制热开关。
- 2) 写时间、环境温度和状态到LCD。
- 3) 等待按键

如果按键被按下，等待弹起周期并再次检查。
- 4) 分析按键。
 - (a) 如果关闭命令被发送，则立即关闭操作单元。
 - (b) 如果周期选择命令被发送，则改换到下一周期记录。
 - (c) 如果时间设置命令被发送，则调整在当前周期记录里的时间。
 - (d) 如果温度设置命令被发送，则调整在当前周期记录里的温度。

2.3.3 流程图

图2-1基本上是嵌入式软件里主要和次要任务之间关系的一种表达。此流程图帮助决定：

- 什么功能出现在哪个逻辑模块里。
- 希望由函数库来支持哪些功能

也能够对流程图的重要结构给出标识符。

2.3.4 状态图

软件可能用来表达在处理外部交互或内部事件后在状态间转换的不同状态。图2-2显示了这些状态和使它们变化的相应激励条件。

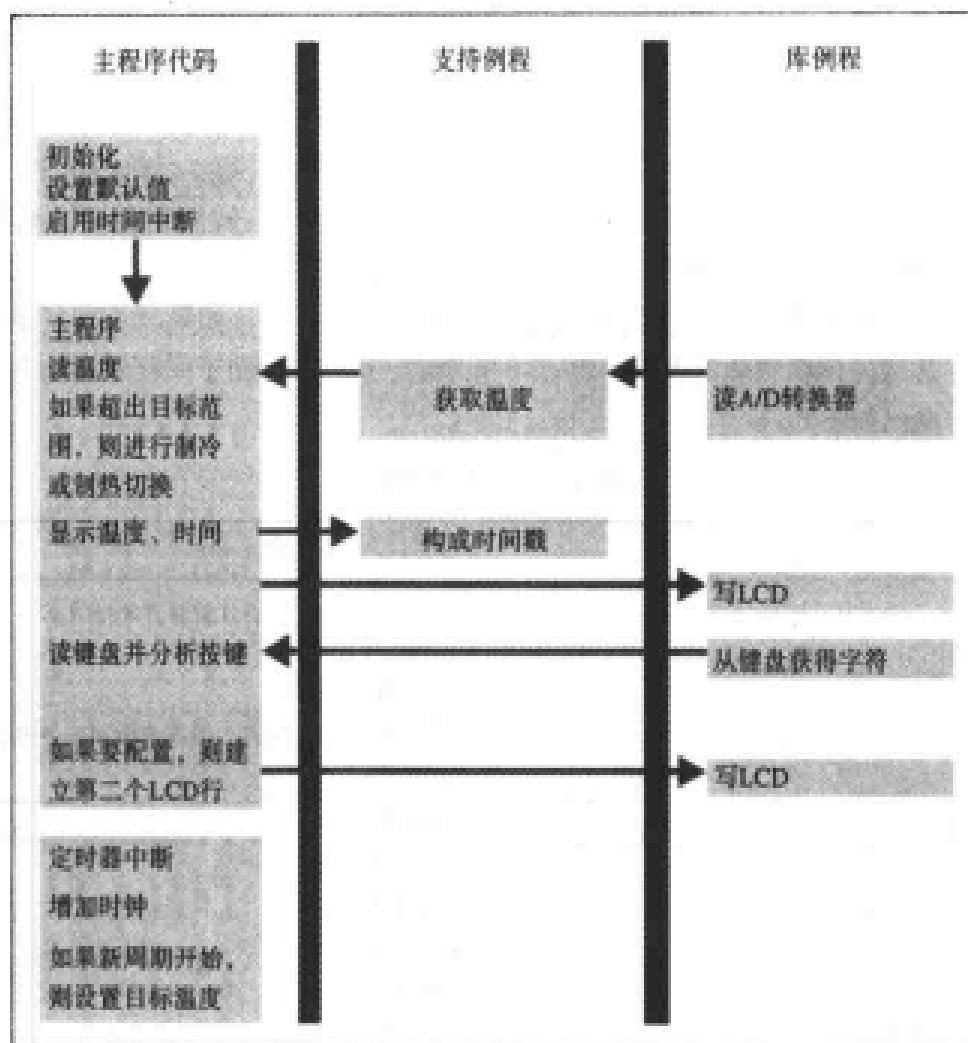


图2-1 关于算法的数据流程

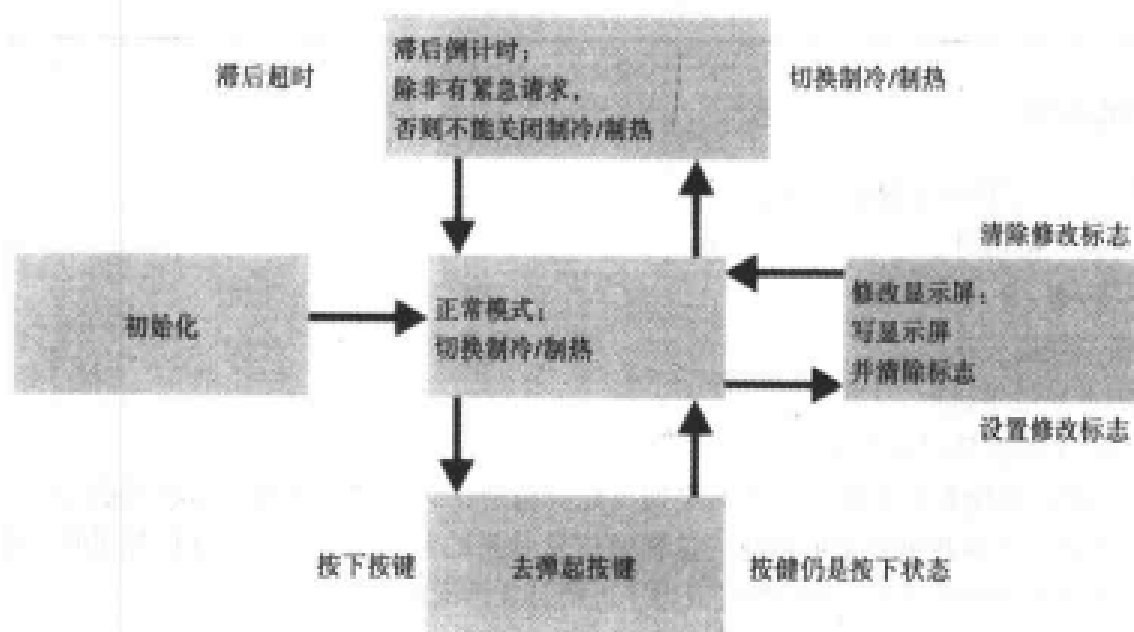


图2-2 关于算法的状态图

2.3.5 资源管理

在微控制器的受限环境里，一个多余的变量或常量能够改变被选择芯片的内存需求，进而影响到价格。像多语言支持这样的特征能够迅速地使资源需求膨胀到一个新的水平。

显式地规划出必需的资源是明智之举。这不是极其幼稚的——我们还在这里谈论通用变量，而不是像页面0访问、串行ROM或者其他技术选择那样的具体细节。

如果您以前写过汇编语言程序，则估计内存需求会更容易些。如果没有那样的经验，写样本代码并编译它是通向精确预估的惟一道路。幸运地是，采用C有助于保存所有的开发成果。

一个粗略的概况如下。

表2-2 估计内存需求

变量/模块	资 源
实时时钟	~10字节RAM，用于计数器和文本表示
日常周期记录	~20字节RAM
用户设置	~10字节RAM
栈	~10字节RAM：2或3个函数调用和一个中断
局部变量	~10字节RAM
RAM估计合计	~60字节RAM
常量	~100字节ROM
中断服务例程	~100字节ROM
初始化	~50字节ROM
主程序	~300字节ROM
A/D转换（温度传感器）	~50字节ROM
LCD	~300字节ROM，依赖于接口类型而具有较大变化
键盘解码	~100字节ROM
ROM估计合计	~1000字节ROM

2.4 测试规划

调试嵌入式软件的建议步骤如下：

- 针对调试而设计。
- 代码审查。
- 在模拟环境里执行。
- 在仿真环境里执行。
- 按测试套件遴选目标系统。

硬件和软件都能得益于调试需要的早期考虑。特别在具有字母数字显示屏的系统里，软件能够通知错误或其他超出规范的信息。这样的信息对测试者和最终用户都是很有用的，但是如果市场不能容忍有错的设备，则软件可以用于证明可靠性问题。

在没有显示面板的情况下，LED能够发送有意义的状态或者事件信号。运行时诊断反馈信息、应该在伪代码和资源管理中体现。

调试的第1步，需要检查由编译器产生的汇编代码。在8位CPU上的嵌入式控制应用程序是足够小的，体系结构也足够简单，开发者能够很容易地审查产生的全部汇编语言代码。按行排列C源代码片段与它们产生的汇编代码的清单文件提供了最容易浏览的形式。

然而，经过这一步之后，测试变成一个挑战：当正测试中的代码实现了机器的最基本行为时，深入系统里的调试变得更加困难。一个故障可能彻底阻止来自嵌入式系统的任何有意义的响应，而桌面操作系统却能够提供内核卸载或其他诊断帮助。

为使深入系统里的调试成为可能，要将模拟器和仿真器与嵌入式系统相匹配。每种工具尝试模拟目标环境的不同部分，从而可以彻底和容易地检测软件性能。在不需要或不关心硬件的条件下，纯软件的模拟器最适合用于检查算法性能和正确性。在真实世界里，仿真器更加集中于I/O和内部的外围设备操作。您将需要访问至少一个仿真器。因为工具选择是与硬件设计过程和处理器选择紧密联系的，所以对它做了讨论。

最后，在测试套件里放入一个原型设备能提供工作软件的最准确证明。

结果

我们的设计将有一个LCD面板。由于这项能力，系统能够写调试消息到显示屏。这些消息可能包括打开电源时的一个回应按键或者显示状态信息的“耀眼屏幕”。

编译器必须支持调试。产生的汇编代码必须是可用于检查的。

选择的产品应该为模拟器执行源代码级的调试提供方便，能匹配当前正执行的机器代码与原始C代码。对于自动调温器而言，模拟速度不是一个关键因素，惟一的时间依赖的功能是实时时钟。

测试套件由一个灯泡和风扇组成，它们由控制器切换开关并指向热敏电阻。这是一个最简单有效的方案。

第3章 微控制器



本章回顾微控制器特征并概述在8位微控制器市场里的可能选择。某些特征可能在中央处理器里看到过的，如图形增强或浮点支持等，在嵌入式系统里是不存在的。

计算机硬件设计最令人注意和具有超凡魅力的部分是中央处理单元的选择。在桌面世界里，处理器选择围绕着同Intel x86产品线的兼容性展开：与Intel兼容的处理器，接近兼容的处理器和完全不兼容的处理器。

在嵌入式世界里，没有这样的连续性，特别在谈论一个新的设计时。8位控制器市场竞争非常激烈，主要是因为它在容量上的焦点。它通常没有商标名称识别，且消费品生产厂商不愿意让用户知道技术细节。如果用户关心驱动他们产品的芯片，他们则可能要去了解超越产品预期应用的新应用领域。

8位微控制器不像32位处理器那样是程序员友好的。32位处理器高度优化的体系结构的后天增强，如额外的ROM地址空间，能够快速超出8位体系结构的限制。这反过来推动处理器设计者在组装机里增加如存储体切换或者寻址限制等技术以作为补偿。

最后，如体系结构的预期寿命等因素也应该考虑到。首选处理器达到它的产品生命周期的末期的时候，采用产生设备程序的C编译器可以减少不断改换处理器的成本。

8位微控制器具有计算机的所有传统功能部件。

中央处理单元（CPU） 微控制器的算法和逻辑单元针对存在于这么小的体系结构里的有限资源受到限制和优化。多路复用和分路操作非常罕见，并且浮点也不存在。寻址模式有时受到令人恼怒的限制。

ROM和RAM 8位微控制器很少有ROM和RAM寻址超过16线的（64 Kb）。如果某个芯片封装暴露所有地址或数据总线，则它们提供的寻址空间只有几千字节。MCU（微控制器单元）包含少量的内部RAM和ROM阵列。因为程序化单个芯片的需要，ROM往往像电子可编程（或电子可擦写）内存一样常见。

定时器 有两种常见类型：计数器和监视定时器。简单的计数器能够对一个时钟周期或者输入信号作出响应。在到达一个零点或者一预置的阈值时，它们能触发一个中断。

中断电路 在通用微处理器有多个通用化的中断输入或者级别的场合，一个微控制器有多个专用于特殊任务的中断信号：计数器超时，或者在某个输入针脚上的信号变化。

那就是说，上述情况是在控制器根本有中断的条件下；如果预期的应用足够简单而不需要它们，就不能保证设计者将一定包含它们。

输入与输出 大多数芯片提供能够切换外围设备的一些I/O线。偶然地，这些针脚能够减弱大电流以减少外部组件。某些变种提供A/D和D/A转换器或者驱动某些设备（如红外LED）的特殊逻辑。

外设总线 并行外设总线将减少“单片机”的优势，因此它们的应用是受到妨碍的。因为速

度在嵌入式系统设计里不是位于前列的功能要素，几个有竞争力的串行外设总线标准已经开发出来。仅仅使用1~3根线，这些总线可使外围设备芯片，例如ROM，与微控制器交互而不独占现有的接口线路。

微控制器的微小尺寸的主要后果是，它的资源相对于桌面个人计算机按比例限制了。尽管具备了计算机的所有特征——RAM、ROM、I/O以及微处理器——但是，开发者不能指望具有其上有8个并行位的I/O端口。

在决定首选处理器之前，必须考虑针对目标可得到的外部开发工具。一个嵌入式系统不像一台PC机那样是自运行的。为开发嵌入式软件，开发工具必须运行在一个桌面计算机上，并至少使用某些非常特殊的硬件。

3.1 中央处理单元

各种不同微控制器中，寄存器的数量和名称可能不同。它们有时出现在某个内存地址空间里，有时又完全各自独立。尽管名称可能不同，但是某些寄存器对大部分微控制器是通用的。

- 累加器
- 变址寄存器
- 栈指针
- 程序计数器
- 处理器状态寄存器

用C直接访问累加器和变址寄存器只能偶尔令人满意。C的register数据类型修饰符等于向一个寄存器发出的要直接访问的“请求”：如果编译器不能令人满意地做到这点的话，实际上则不使用寄存器。

然而，当使用寄存器是最合适的或者是必须的时候，另一类型声明能够链接变量名和一个寄存器本身。Byte Craft编译器提供了registera类型（以及其他寄存器的相应类型）。给一个registera变量赋值产生进入到累加器寄存器的一个数据装载，但是不产生到内存的一个存储。对该标识符的求值返回在寄存器里的值，而不是来自内存的值。

```
registera important_variable = 0x55 ;
```

直接访问栈指针或程序计数器是不太明智的。采用C语言的目的是从直接机器语言引用中抽象程序逻辑。函数调用和循环将使设备依赖的栈操作和分支操作平衡，是结构化编程的最好方法。如果有必要，可以对带标号的目标使用C关键词goto：编译器将插入合适的跳转指令，并且，最重要地是，自动地考虑任何分页或设置问题。

3.1.1 指令集

期待出现在通用微处理器单元MPU（microprocessor unit）里的机器指令有：乘法、除法、表查询或者乘法-加法，而在8位处理器里的相应指令并不总是出现在控制器家族的每个变种里。

一个#pragma语句能够通知编译器目标芯片确实有一个可选指令功能，因此编译器能够利用该指令提供的便利条件而优化代码。这样的例子出现在MC68HC05C8的头文件中。

清单3-1 指令集配置

```
#pragma has MUL;
#pragma has WAIT;
#pragma has STOP;
```

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

3.1.2 栈

如果处理器在通用内存里支持栈，则记录该栈所必需的空间是从RAM分配的，否则该空间用于全局变量。不是所有的栈都记录在主（或者数据）内存；Microchip PIC和Scenix SX体系结构使用在用户RAM外的栈空间。

检查由函数调用和中断而存储的返回信息的深度是很重要的。编译器可能报告栈溢出（意味着栈太小了），但是栈声明也可能比必需的要大。

除了声明一个内存区域作为栈的预留区外，没有任何其他需要担心的了。考虑下面Motorola MC68HC705C8栈。该栈从地址00C0到00FF有64字节。

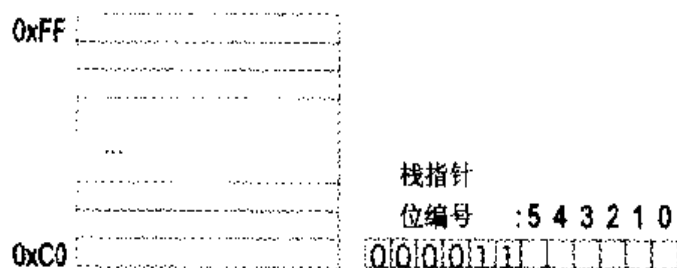


图3-1 MC68HC705C8 栈

在C中，必须做如下声明：

```
#pragma memory stack [0x40] @0xFF;
```

因为栈大小和配置将是随处理器族的不同而改变的（或者甚至在相同族里的变种之间也是不同的），该声明使编译器精确地意识到有多少可以得到的空间。如果不需要64字节，则可减少其大小，把0x40变为更小的数即可。

编译器能够提供函数调用的更深层次的信息，参见9.6节的CALLMAP选项。

3.2 内存寻址和类型

多数小的微控制器几乎不提供RAM。对于桌面应用程序员而言，由完全消耗掉RAM或者ROM引发的那种惧怕的感觉是新奇的。除了对失败的内存分配进行粗略的检查，程序员依靠成兆的RAM和交换文件几乎能够完全避免内存不足的错误。

C编译器支持重用内存，无论什么情况下都可能支持。编译器为了在多个局部范围里重用内存，不断判定哪个时间里哪些位置是空闲的。当然，“空闲”意味着只有被下一个函数调用重新初始化，它才是被该子例程可读的。

因为内存问题，某些经典的程序设计技术超出了8位微控制器的能力。重入的或者递归函

数，即那些在桌面系统程序设计里的精华，是假定有足够的栈空间，但在嵌入式环境里却是不可能的。

3.2.1 RAM和ROM

RAM和ROM在微控制器里是永久性地被划分的。它们可能是不同地址空间的部分。

控制器的所有存储空间减去RAM或者ROM所占有的空间，余下的是未实现的内存地址空间部分。指令取或读或写到这些区域能够导致不可预测的或者错误的结果。

声明可用的RAM和ROM，指示编译器哪里能够安全地放置程序代码或者数据。Byte Craft编译器需要所有的内存资源都被声明。这些声明能够简单地声明可得到的内存类型、大小和位置，或者可选地给区域指派一个符号名称。

命名地址空间在优化处理之外带给程序员一些控制。如果处理器具有更快访问部分内存的能力（例如，在680x上的page 0），并且头脑中有一个特别的规划，那么声明将存留于那个内存区域的变量。

清单3-2 在命名地址空间里声明

```
#pragma memory ROM [0x4000] @ 0xA000;
#pragma memory RAM page0 [0xFF] @ 0x00;
#pragma memory RAM page1 [0xFF] @ 0x100;

/* ... */

/* my_variable将出现在page0。如果处理器有访问page0的特殊指令。编译器将产生它们以
便赋值和后续引用。 */

int page0 my_variable = 0x55;
```

3.2.2 ROM和程序设计

在制作一个封装的ROM产品之前，可编程ROM或PROM开始是作为一个昂贵的工具来原型化和测试应用代码。近几年来，PROM已相当流行，许多开发者考虑将它作为大规模生产芯片中封装ROM的出众替代品。

随着微控制器应用变得越来越个性化和越来越复杂，开始需要对它们进行维护和支持。许多开发者采用PROM设备为客户提供软件修改，而免除了发布新硬件的成本。

可编程ROM的分类在下面描述。

Fused ROM是传统的PROM，具有已编程的ROM单元，采用的方法是在一个存储体阵列（即ROM单元）里，按照比特模式有选择地烧断熔丝。可编程只有在外设设备支持下才能实现。

EPROM（可擦写可编程ROM）是非易失性的并且是只读的。它必须通过暴露在紫外线里来擦除。

EEPROM（电子可擦写可编程ROM）设备有超过EPROM设备的重要优势，它们允许有选

择地擦写内存段。EEPROM最平常的使用是记录和维护对应用至关重要的配置数据。例如，调制解调器采用EEPROM存储体记录当前配置设置。

闪存是EEPROM和EPROM技术之间的一个经济的折中选择。产品有一个基于ROM的配置内核，以及写入闪存的应用程序代码。当想要为客户提供附加功能或者维护升级时，硬件能够在现场重新编程而不用安装新的物理芯片。硬件已被置入到配置模式，配置模式传递控制到写入ROM的内核。然后，该内核处理为删除和重写闪存内容所必需的软件步骤。

依赖于目标芯片，EEPROM和闪存在程序控制下是可编程的。电子必须等待电荷传输并慢速工作以免设备过热，因此，这种编程过程要占用一些时间。

3.2.3 冯·诺依曼与哈佛体系结构

冯·诺依曼体系结构有一个单一的公用的内存空间，程序指令和数据都存放在这个空间里。提取指令和数据是通过一个单一的内部数据总线进行的。

哈佛体系结构计算机针对程序指令和数据有分离的内存空间。有两个或更多的数据总线，这就允许同时访问指令和数据。CPU利用程序内存总线获取程序指令。

程序员不必关心他们针对什么体系结构编写程序。C编译器能够补偿他们各自缺点和特异的大部分。以下是某些更通用的特征的解释；用以分析由编译器产生的代码。

- 冯·诺依曼体系结构的机器的代码生成常利用了处理器能够执行RAM外的程序这样一个事实。某些数据类型的操作可能实际上预先准备了与操作代码相关的RAM位置，然后针对它们（RAM地址）进行分支操作！
- 因为哈佛体系结构机器有针对数据的明确内存空间，所以为数据存储而使用程序内存是更加复杂的。例如，一个声明为C常量的数据值必须作为一个常量值存储在ROM。一些芯片有允许从程序内存空间获取信息的特殊指令，这些指令总是比从数据内存获取数据的等价指令更复杂或成本更高。其他简单芯片没有这些指令，数据必须被装载，例如，通过一条返回指令的副作用完成。

3.3 定时器

定时器是按照固定速率的时钟脉冲增加或减少的计数器。通常，一个固定时间间隔引发一个中断：定时器计数到0，溢出到0，或者达到一个目标计数。

在微控制器里，定时器是标志产品竞争性的特征。提高了精度和智能的定时器或者计时单元是很容易得到的。目前，定时器的不同类型给工程师带来了许多选择的空间。

编码预定义标器并启动时钟是软件程序员的任务。只要知道处理器时钟频率，并选择了正确的预置数值，程序员就能够得到正确的定时器时钟周期。

程序员与定时器的接口是几个命名控制寄存器，用`#pragma port`语句声明并作为变量读或者写。

如果定时器中断得到支持，则它能够用一个`#pragma vector`语句声明，进而通过一个作为函数编写的关联的中断服务例程来得到服务。

清单3-3 定时器寄存器和中断处理程序

```
#pragma portr TIMER_LSB @ 0x24;
#pragma portr TIMER_MSB @ 0x25;

#pragma vector TIMER_IRQ @ 0xFFE0;

void TIMER_IRQ(void) {
    /* IRQ处理程序代码 */
}
```



3.3.1 监视定时器

COP（计算机正常操作）或者监视定时器检查代码执行是否失控。通常条件下，监视定时器必须在重置后的头几个循环周期里被打开一次。然后，在执行期间，软件必须周期性地重置监视。

如果处理器执行离开了正常轨道，监视不可能被可靠地重置。这正是那种需要被修补的状态：间接跳转到一个意想不到的地址可能是其原因，发送从来不会被收到的外部信号的循环也是一个可能的原因。

监视超时能够使处理器进入到一个已知的状态，通常是RESET状态，或者执行一个中断。监视定时器的硬件实现在不同处理器间变化相当大。某些监视定时器能够针对不同超时延时编程。

在C语言里，重置监视的序列同赋值给一个端口一样简单。

清单3-4 重置监视定时器

```
#pragma portw WATCHDOG @ 0x26;
#define RESET_WATCHDOG() WATCHDOG = 0xFF

void main(void) {
    while(1) {
        /* ... */
        RESET_WATCHDOG();
    }
}
```

3.3.2 实例

下面是一些样例配置：

- National Semiconductor公司的COP8SAA7有一个称作T1的16位定时器，一个称作T0的16位空闲定时器，以及一个监视定时器。空闲定时器T0帮助维护空闲模式期间的实时和低功耗。定时器T1用于与3种用户可选的模式相关的实时控制任务。

- Motorola公司的MC68HC705C8有一个16位计数器和一个COP监视定时器。COP监视定时器是用户启用的，提供可选择的超时周期，并且可以用两条向COPCR寄存器写的指令来重置它。有意思的是，COP监视是依赖于系统时钟的。如果时钟停止，则时钟监视电路重置MCU，从而致使COP监视无用。
- Microchip公司的PIC17C42a有4个定时器模块，分别称作TMR0、TMR1、TMR2和TMR3，以及一个监视定时器。TMR0是一个具有可编程预置数值的16位定时器，TMR1和TMR2是8位定时器，TMR3是16位定时器。

3.4 中断电路

微控制器通常提供硬件（信号）中断源，有时提供软件（指令）源。在有有限引脚数的封装里，IRQ信号可能没有被暴露或者可能是与其他I/O信号多路复用的。

能够被禁止的中断是可屏蔽中断，不能被禁止的中断是非屏蔽中断。例如，RESET是非屏蔽中断。不管当前正执行的代码，CPU必须立即服务于RESET中断。

中断信号是异步的：它们是能够发生在一个指令周期期间、之后或之前的事件。处理器能够使用两种方法之一确认中断：同步或者异步确认。

大部分处理器同步确认中断：它们在处理中断之前处理当前的指令。相反，异步确认的处理器停止当前指令的执行转而服务于该中断。虽然异步确认比同步确认更及时，但它带来了中断代码干扰已进行中的指令的可能性。

例如，一个中断例程要修改一个主程序代码正读取的各字节数值。如果主程序代码在一个多字节提取操作里读该值时被半道中断，则已装载的值在没有任何提示下变成无意义的值。

在中断和主程序代码之间单程读和写变量的代码应遵从我们的建议（参见4.4.2节）。为提供完整性保护，编译器需要采用不可分割指令，或者临时禁止中断，以保护主程序代码。

同步确认也不是完美的方案。当一个多字节操作需要几个指令时，以上同样的问题也会影响同步确认的处理器。

3.4.1 向量和非向量仲裁

微处理器服务于中断有两种竞争的方法。向量仲裁需要指向中断服务例程（ISR）的指针表。非向量仲裁期待ISR的第一条指令位于一个预定义的入口点。多数8位微控制器采用向量仲裁中断。

当编译器为ISR产生代码时，它把开始地址放到ROM映像里特定的中断向量里，或者重定位ROM里入口点处的代码。编译器也可能自动产生仲裁代码：在评估ROM使用时，记住检查它。

当一个中断发生时，处理器将禁止中断以防止服务例程被其自身中断。然后，一个向量机器读取包含在特定中断向量里的地址，它跳转到那个地址并开始执行ISR代码。

相反，一个非向量系统简单地跳转到已知的起始位置并执行那里的代码。为了实现优先级，ISR可能不得不按顺序测试每个中断源，或者简单跳转到ISR驻留的主体所在的不同位置。

因为在非向量系统里需要额外的处理，所以向量中断更快一些。总之，非向量ISR在少于5个中断的微控制器里是可行的。

表3-1显示了8位微控制器的主流族的仲裁方法。

表3-1 中断仲裁方法

体系结构	仲裁	注 释
Motorola 6805/08	向量	向量位于支持内存的顶端
National COP8	混合	参见本表之后的文字
Microchip PIC	非向量	某些模式没有中断，而有些为中断组提供向量派遣
Zilog Z8	向量	必须设置优先级
Scenix SX	非向量	无优先级
Intel 8051	非向量	每个中断跳到一个不同的、固定的ISR入口点
Cypress M8	非向量	对于每个中断，处理器跳到一个不同的、固定的ISR入口点。它们被称作“向量”且有两字节长。为正确跳转到ISR，在那些位置必须有一条JMP指令

National Semiconductor公司的COP8使用了一种混合方法，所有中断按照非向量方式转移到一个公共位置。在那个位置，代码要么执行VIS指令，该指令在多个活动中断源里仲裁并跳转到从向量表里得到的地址处；要么显式地投送中断条件给系统并按照用户定义的方式处理它。后一种方法也许是有用的，但是也有许多缺点。

表3-2显示了COP8向量表，它是COP8SAA7设备所必需的。排列顺序是VIS指令的实施顺序。

表3-2 COP8向量中断

序 号	中 断 源	描 述	向 量 地 址 ^①
1	软件	INTR 指令	0bFE - 0bFF
2	保留	为将来保留	0bFC - 0bFD
3	外部	G0	0bFA - 0bFB
4	定时器 T0	下溢	0bF8 - 0bF9
5	定时器 T1	T1A/下溢	0bF6 - 0bF7
6	定时器 T1	T1B	0bF4 - 0bF5
7	MICROW-IRE/PLUS	BUSY Low	0bF2 - 0bF3
8	保留	为将来保留	0bF0 - 0bF1
9	保留	为将来保留	0bEE - 0bEF
10	保留	为将来保留	0bEC - 0bED
11	保留	为将来保留	0bEA - 0bEB
12	保留	为将来保留	0bE8 - 0bE9
13	保留	为将来保留	0bE6 - 0bE7
14	保留	为将来保留	0bE4 - 0bE5
15	端口 L/唤醒	端口 L 边界	0bE2 - 0bE3
16	默认	无中断时，执行VIS指令	0bE0 - 0bE1

① b代表VIS块。VIS和向量表必须在相同的256字节的块里。如果VIS是一个块的最后地址，则该表必须在下一块里。

3.4.2 中断期间保存状态

对所有芯片而言，中断处理都保存机器的一个最小处理器状态，通常为当前程序计数器。



这是为了确保在中断服务之后，主程序将从正确的地方继续执行。

除此以外，机器状态保存变化很大。在任何情况下，都是由程序员提供代码，来保存尽可能多的额外状态。通常，每个中断处理程序将在尝试做任何其他事情之前完成状态保存的工作。被保存状态信息的位置和可访问能力是随机器的不同而变化的。

表3-3 中断期间的处理器状态保存

体系结构	中断栈操作行为
Motorola 6808	除了栈指针的高字节，所有寄存器被自动保存和恢复
Motorola 6805	所有寄存器被自动保存和恢复
National COP8	程序计数器（PC）被压进栈
Microchip PIC	PC被压进栈
Zilog Z8	PC和标志被压进栈
Scenix SX	PC被压进栈，其他寄存器被映像
Cypress M8	PC和标志被压到程序栈上

许多C编译器在数据内存里为内部使用，如伪寄存器，预留某些位置。编译器文档应该描述必须写什么代码以便保留位于这些内存块的信息。如果编译器为16位数学操作创建了一个伪寄存器，而中断处理程序并不执行更改这个伪寄存器的16位操作，则可不必要保存它的状态。

3.4.3 执行中断处理程序

为最小化中断例程被自身中断的可能性，微控制器在执行中断处理程序时将禁止中断。

在主程序代码的定时临界区，手工屏蔽中断是有用的。这样做的可能性是由设计决定的，用C语言实现它也很容易。不用花费更多的精力就可以归纳这个过程。

就Byte Craft编译器而言，在一个头文件里的一些简单宏能够创建合适的指令。下面的代码使用由编译器本身定义的符号来选择合适的指令。

清单3-5 跨平台中断控制指令

```
#ifndef CYC
#define IRQ_OFF() #asm < DI>
#define IRQ_ON() #asm < EI>
#endif

#ifdef C0F8C
#define IRQ_OFF() PSW.GIE = 0
#define IRQ_ON() PSW.GIE = 1
#endif

#ifdef C6805
#define IRQ_OFF() CC.I = 0
#define IRQ_ON() CC.I = 1
#endif
```

3.4.4 多个中断

对于某些机器，CPU首先取来和执行一条程序指令，然后检查悬挂的中断。这保证不管有多少中断在排队，机器将总是进入程序代码：在每个主程序指令之间将至多执行一个中断处理程序。

对于大多数机器，CPU在取下一条指令之前将检查中断。只要控制器侦测到一个悬挂的中断，它就在取下一条指令前服务这个中断。这意味着连续不断的悬挂中断可能使主程序得不到执行。另一方面，它保证中断在任何主程序代码执行前得到服务。这一信息对调试是很重要的：它能够帮助解释为什么主程序软件不响应了。

CPU怎样决定哪个中断首先应该服务？如果两个中断在同一时刻发出信号，则应该由硬件优先级来决定。

3.4.5 RESET

一些简单芯片不支持中断，RESET序列除外。如果其预期的应用只需要一个简单的轮询循环，或者根本没有任何输入，则对于额外硬件没有需求。

惟一通用的中断信号是RESET。RESET产生的原因如下：

- 初始加电。
- 手工重置（在一个外部RESET针脚上有信号）。
- 监视超时；
- 低电压，如果芯片支持后备电源监视的话。
- 从一个非法或者未实现的地址取指令，如果芯片实现这类保护的话。

RESET中断迫使芯片的行为就像在电源关闭后又快速打开一样。既然它不是实际地开关芯片电源，则易失性内存、I/O端口或者处理器寄存器的内容仍保持原封不动。

对这个特征的利用很复杂，但是是可能的。如果编译器支持用户编写的初始化函数，则能够检查内存里的特殊值，并决定载入默认值或者不载入。这可用于检查RESET是冷的（电源被关闭过——采用默认值）还要热的（电源没有被关闭过：保护没有受到影响的数据）。

还有干扰这个策略的情况。在监视超时的情况下，数据在物理上是合法的（与监视RESET以前相同），但在逻辑上是存在问题的。

3.5 I/O端口

输入/输出信号允许微控制器控制和读取继电器、灯泡、开关或者任何其他具体设备。更复杂的组件，如键盘、LCD显示屏或者传感器，也能够通过端口访问到。本节谈论标准I/O的程序设计问题。更特殊的外设，像A/D转换器和通信总线，将在下一节讨论。

端口通常由8个排列在1字节大小的I/O数据寄存器里的可切换电路组成。如果一个端口同时具有输入和输出功能，则它同时也具有相应的寄存器，用以指定端口（或者端口的每个位）的操作方式。在许多设备里，这类寄存器被称作数据方向寄存器DDR（Data Direction Register）。

端口经常支持三态逻辑。三态增加了除输入和输出外的第3种有用配置：高阻抗。高阻抗模

式是一种未定义的或者漂浮不定的状态。在那个时刻，可以当作该端口实际不是电路的一部分。

因为微控制器希望尽可能地代替多个设备，所以端口经常包括额外的功能，如内部上推或者下拉等。这些电气特征免除了某些干扰。

数据方向、三态控制和可选的上推或者下拉都是需要程序员来控制的。如同桌面计算机系统那样，端口和它们的控制寄存器作为内存单元或者特殊的I/O寄存器出现。

下面列出了一些端口配置的例子。

- COP8SAA7具有称作C、G、L和F的4个双向8位I/O端口，每个端口的每个位都可以是输入、输出或三态之一。针对每个端口的程序设计接口有一个相应的配置寄存器（决定端口的行为）和数据寄存器（为端口提供数据或者接受来自端口的数据）。
- Motorola MC68HC705C8有3个8位端口，分别称作A、B和C，每个端口依赖于DDR的值，要么是输入，要么是输出。还有一个用于串行程序设计具有7位的固定输入端口，叫作端口D。
- Microchip PIC16C74有5个端口：PORTA到PORTE。每个端口具有一个相应的控制数据方向的TRIS寄存器。PORTA使用寄存器ADCON1选择模拟或者数字配置。PORTD和PORTE能够配置为8位并行伺服端口。

端口和它们相应的配置寄存器不是RAM内存单元，它们在电气特性上是不同的。如果不是开发商显式许可，读或写端口可能是非法的或危险的。通过在端口声明中指定可接受的模式，编译器能够注意到不正确的读或写。

在Byte Craft编译器里，可以使用#pragma语句向编译器声明端口。

```
#pragma portrw PORTA @ 0x00 ;  
#pragma portw PORTA_DDR @ 0x04 ;
```

可接受模式的规范为：portr是可读的，portw是可写的，portrw是可读写的。

模/数转换

需要经常把外部模拟信号转换为数字信号，或者把数字信号转换为模拟信号。A/D或者D/A转换器执行这个功能。

转换背后的技术以及像自动检测仪表或语音处理之类的模拟规范的竞争环境，决定了有许多种转换的途径，需要权衡正确性、精度和时间。

典型地，A/D或D/A转换器的支持函数库是封装成C函数库的首要候选对象。需要着重指出的是，转换过程也许要占用一些时间。

Byte Craft编译器将以两种方式支持这种类型的外设。

- 在设备头文件里用#pragma port声明控制端口。
- 声明一个中断并用一个ISR函数服务它，该中断是使用#pragma vector语句通过转换外设引发的。这是一种直观的处理转换的方法，但需要较长的时间。

多数微控制器在A/D转换中使用逐次逼近转换器（successive approximation converter）。转换器从MSB（最高有效位）开始一次转换一个比特并决定下一步是更高位还是更低位。这种技术速度很慢，并且要消耗大量的功率。同时，它是开销低的，且具有一致的转换次数。

Microchip PIC16C74有一个带8个模拟输入的A/D转换器模块。这8个输入被多路复用为一个抽样-保持,即转换器的一个输入。

线性倾斜转换器(single slope converter)出现在National Semiconductor公司的COP888EK芯片里。它包括一个带有输入捕获和恒定电流源功能的模拟MUX/比较器/定时器。其转换时间变化很大并且相当慢。同时,它具有14位到16位的精度。

快速转换器(flash converter)检查每个标准并决定电压标准。它是非常快的,但要绘制大量的电流曲线,因此不能超出10位。

3.6 串行外设总线

充足引脚数的单片机微控制器能够向外部暴露地址、数据和控制信号,但是,这就消除了单片机设计的好处。

串行外设通信的标准有好几个。一般芯片采用1~3根外部电线与1个或者多个外设通信。

当然,串行化频繁的ROM或RAM访问会影响执行速度。串行外设没有包含在处理器的寻址范围里,所以串行编程ROM是不可能的。

编译器通过使串行外设的数据访问更直观来提供支持。Byte Craft编译器提供SPECIAL内存声明。使用它能够在内存映像里声明远程设备的寄存器或者内存,而编译器能够理解它。然后,编写设备驱动例程以读和写每个SPECIAL内存区域。

访问SPECIAL内存区域声明的变量或者端口要接受特别的处理。读取一个SPECIAL变量的值将执行相应的读例程,其返回值就是读取的值。赋一个新值给SPECIAL变量即传递该值给一个相应的写例程。读和写例程能够指导外设总线事务以获取或设置变量的值。

总线标准和驱动器例程是函数库实现的主要目标。

表3-4 串行外设总线选项

标 准	生 产 商	注 释
PC	Philips	在2根线上操作的同步串行外设接口。这2根线由串行数据线和串行时钟线组成,都是双向的。没有指定程序设计接口。
SCI	多个	板级串行通信的增强型UART。通过2根线进行异步操作。
SPI	多个	在4根线上操作的同步串行外设接口:SPI时钟(SCK),MOSI(从入主出)MISO(从出主入)和slave选择(SS)。生产商重贴标签,或增强了该标准。例如,National Semiconductor公司提供相似的(可能兼容的)MICROWIRE/PLUS设备。

3.7 微控制器的开发工具

用C开发软件需要使用一个桌面计算机,以运行交叉编译器。从而就能够按照下面的方法之一编写程序或者评估目标系统。

手工程序设计 开发者对EEPROM微控制器编程,并在每次重复测试后替代目标里的程序。这是耗费时间和精力的事,但提供了最真实的测试环境。其结果不会受到测试工具的不良影响。

模拟器 开发者载入目标代码到模拟最终环境的软件程序里。这种安排最适合于检测复杂程

序设计。

仿真器 开发者把设计方案里的微控制器（或者像编程ROM那样的一个外部芯片）用一个特殊的硬件替代，由它在提供到开发平台的链接的同时并仿真微控制器设备。一个设计良好的仿真器对于目标系统而言，比起正常微控制器来没有什么两样，但是，却允许用户同时观察微控制器的行为和检测目标平台的硬件。

开发工具是处理器选择时要考虑的一个因素。编译器能够产生信息以使原始源代码与模拟器或仿真器使用的目标代码链接。留心一下与您的编译器兼容的产品。



第4章 设计过程

设计过程反映问题规范，该过程要对以前提出的每个常规问题做出具体的决定。

4.1 产品功能

我们能够反映产品需求，即面向用户的检查清单，该清单描述产品将要执行的任务，以及将要被设计的设备的某些细节。

结果

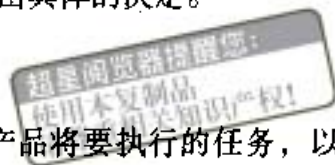
- 程序测量当前温度。我们将不得不服务和读取连接到热敏电阻的A/D转换器。为使芯片数量最小化，A/D转换器将是十分基本的。
- 程序将按24小时时钟计实际的时间。利用每秒钟一次的中断能够计分钟和小时。我们不会受到日/日期计算的影响——不必自动调整夏令时，也不必计算年的问题。
- 程序将接受当前时间设置并重新设置时钟计数。库例程帮助转换内部时钟表达为可显示的格式。
- 程序接受和存储用户选择的制冷和制热温度设置，以及三个日常使用周期的时间设置。以合理的默认值构建，并在RAM里保存当前的配置。如果掉电了，设备不会使任何人处于危险中。
- 程序比较当前温度与当前时间周期的设置，然后根据需要打开或者关闭外部制冷和制热单元。这需要输出一行说明对制冷和制热启动一个继电器。
- 程序防止外部单元在一个短的时间周期里两次改变状态以避免抖动。这就是说，程序在切换操作时保持一个单独的5秒等待时间。立即关机将使这个计数无效。
- 程序在任何时间接受手工修改，并立即关闭所有活动的外部单元。无论键盘是轮询驱动的还是中断驱动的，关机用的一个或两个键应该立即得到响应。

4.2 硬件设计

正如前面提到的，硬件是本书范围之外的内容。我们在这里包括硬件信息，是为了证明我们在设计自动调温器时做出的选择是合适的。

选定的芯片是MC68705J1A，主要是因为其简单性和引脚数少。它刚好具有足够控制所有设备的引脚。

- 14个I/O引脚，外加一个已禁止的IRQ输入。
- 8个对应键盘的引脚（端口a）。
- 2个对应热敏电阻的引脚（1个源于端口b，1个源于已禁止的IRQ输入）。
- 7个对应串行LCD面板的引脚（3个源于端口b，4个源于端口a）。



- 2个对应制冷和制热切换的针脚（端口b）。

J1A是惟一需要的芯片，余下的是具体的部件。

一旦安排好了硬件，任务就转向到设计程序了。

4.3 软件设计

4.3.1 软件体系结构

如前面所述，我们将采用C语言。

预打包的微控制器函数库是由嵌入式目标的特定C编译器提供的，但是它们无论如何也不能像通用计算机程序员所面对的那些函数库那样具有通用性。

针对微控制器的函数库总是提供源代码的。既然最终产品的安全成为像机器控制那样的应用的一个重要因素，那么函数库必须同程序的其他部分一样进行仔细地检查。

为保持生产性，编译器和仿真环境应同扩展的调试信息保持相同的格式。这就允许仿真器执行清单文件的源码级调试。

而在传统情况下，链接器不是严格必需的。

这里不详细讨论开发环境。如果实现版本控制和建立自动化是必须的话，关于配置管理的文本能够对这两个功能提供最好的支持。

结果

编译器将采用Byte Craft有限公司的C6805代码开发系统（CDS）。它产生Motorola、Intel和芯片专用的二进制代码，以及一个在原始源代码旁边放置产生的对应汇编代码的清单文件。

有了Byte Craft CDS，有关设备的特殊细节在一个头文件里放置着，它使用通用标识符来表示这些细节。要保证有设备头文件05j1a.h。当使用EEPROM芯片时，使用文件705j1a.h。如果要改变目标芯片，只要简单改变头文件即可。

用于自动调温器的函数库包含下面的内容。

stdio 包括从显示屏和键盘获取和输出字符串的例程。这个库依赖于其他库做实际的输入和输出。

lcd 包括清除显示屏、移动硬件光标和写字符与字符串的例程。

keypad 包括检查按键和解码按键的例程。

port 提供到j1a芯片的2个并行端口的透明访问。

delay 计时同LCD显示的通信，并弹起键盘。

我们将写一个完整的新库。

timestmp 转换秒数为人可读的时间并做相反的转变。

时钟/计数器中断计算时间，请求显示屏更新，并设置目标温度。主程序实现一个循环以更新LCD、查询键盘、抽样环境温度和切换HVAC机械。

4.3.2 流程图

现在是向算法的流程图增加具体信息的时候了，接下来还将利用它来规划我们的源文件。

结果

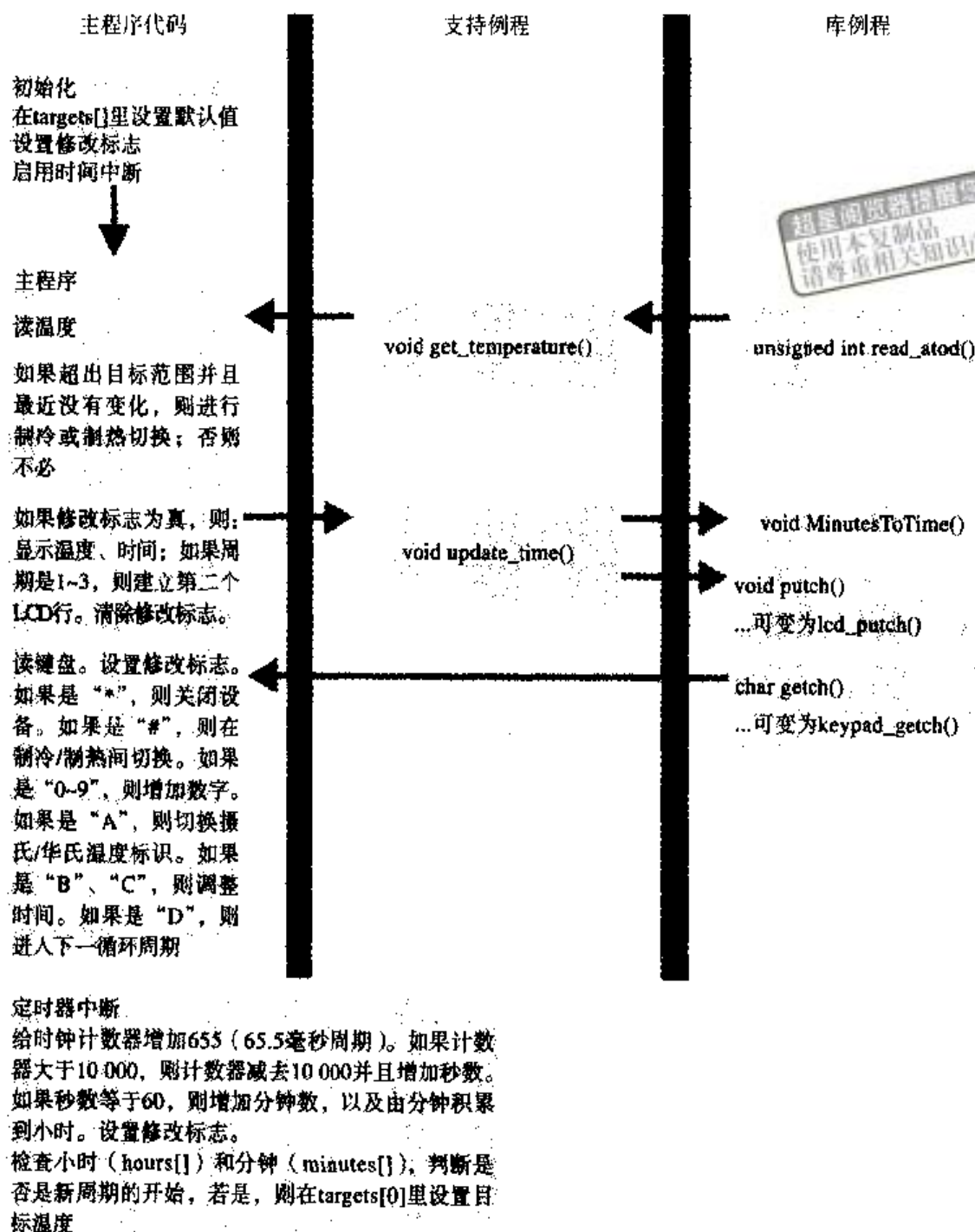


图4-1 关于算法的数据流程(修订版)

4.4 资源管理

既然有了一些关于目标平台、开发软件和数据在软件各部分间流动的方式等的具体信息,下面就能够着手规划资源使用了。

4.4.1 暂存缓冲器

许多C编译器为内部目的如伪寄存器等使用某些可用的RAM。高效的C编译器将支持数据内存中的暂存缓冲器。一个暂存缓冲器是一块能够用于多个目的的内存。一个伪寄存器是一个用于较大数据类型执行基本操作的变量。编译器文档将详细描述暂存缓冲器分配的大小和目的。

例如，如果尝试在一个没有固有16位寄存器的芯片上进行16位数学操作，编译器将专门留出一定比例的RAM作为16位伪寄存器，以便在数学操作期间存储数值。

如果暂存缓冲器分配滥用了内存预算，可以自己考虑重用内存，惟一的前提是必须自己管理变量域。

例如，Byte Craft编译器创立了一个16位的伪变址寄存器_longIX。可用下面的语句重用这个16位地址。

```
long int myTemp @_longIX;
```

如果用myTemp存储一个数值，然后发出一个新的库调用，则这个库软件将禁止执行任何long操作，否则数据将被覆盖。

4.4.2 中断计划

除非您已经研究过驱动程序或者其他低层软件开发，否则，您可能是同中断隔离的。虽然嵌入式C通过提供直观的方法帮助构造和编写中断处理程序，但还是需要一些忠告。

- 主程序处理器状态将怎样得到保护？处理器寄存器也许已自动保存到一个栈上，或者处理器简单地把它们投影到隐含寄存器。如果能够得到多组寄存器，则可能很容易地把主程序寄存器值交换出去。作为一个最后的手段，可以手工地保存寄存器值，并在从中断返回之前恢复它们。

如果在中断里的计算可能擦除临时寄存器的话，则由编译器数学函数使用的临时寄存器也需要被保护。保护这些寄存器将需要多字节传输例程。这些重复序列在一个频繁调用的中断里能够大大增加开销。

- 为中断预计的任务，包括以前的保存和恢复操作，能够及时完成吗？中断调用的频率以及在它们内部要完成的工作量需要正确估计到。

如果有多于足够的时间完成这些操作，则处理器的速度可以降低以换取元器件的利益。

- 中断例程和主程序代码将怎样交互？除了通过禁止中断保护主程序代码的临界区，还有更广泛的同步冲突需要考虑，特别是全局数据。

一个通用的规则是在一个地方写全局变量——主程序或中断代码——并在另一个地方读它们。如果可能，在中断例程和主程序代码之间只实现一条通信通路。

结果

C6805 CDS创建一个叫作_SPAD的4字节暂存缓冲器，它同时为16位操作创建了2个伪寄存器，它们是_longAC（2字节）和_longIX（4字节）。

C6805具有对局部内存的支持，因此要注意计数器或临时变量分配中的节约措施。

芯片jila有一个软中断，它可以被编译器当作一个快速子例程调用来使用。我们将不这样显

式地使用它，我们将禁止该IRQ输入作为一个空闲输入引脚来使用。

芯片j1a也有一个定时器中断，用它执行定时功能。中断将大约每65毫秒运行一次，所以我们需要保留下面的项目。

毫秒计数器 实际上，毫秒计数器需要额外的精确数位以满足发布的规范，因此将保持1/10毫秒的精度。

秒计数器 我们将按分钟显示时间，因此这个计数器只是作为内部使用。

具有小时和分钟的计数器 我们将在后面对它进行更多的解释。

既然需要外部IRQ引脚作为额外的输入，那么不能使用与端口A的引脚0~3关联的键盘中断功能了。

6805中断引起整个处理器状态被保护：累加器、X寄存器、PC、栈指针和状态码。因此不用为这个中断写代码，而可能需要保护伪寄存器。

4.5 测试选择

4.5.1 调试规划

在处理器选择之后，下面可着手规划一个测试战略。

处理器可能以一个硬件调试接口的形式提供某些帮助。

设计软件时，按函数库分组软件是一个良好的组织技术。通过编写短小的测试程序分别测试每个子系统，而每个子系统使用一个函数库。

模块化测试解决了一个有趣的问题：一个带有LCD显示屏的系统能够显示人可读的状态代码或者其他调试消息。但是，直到LCD显示屏本身是可操作的和可靠的才行，否则它是无用的。

用一个测试程序直接考察LCD显示屏的配置：它是一个更复杂的“黑箱”设备，带有一个4位或8位的接口，以及启用、寄存器-选择和读/写线路，它们必须是依照计时范围发出中断信号的。在我们的设计里，多路复用LCD数据总线和键盘是成本划算的。在您的设计中，LCD总线甚至可能按照更加复杂的方式被复用连接。您需要一个测试程序来驱动为硬件所定制的函数库。

4.5.2 代码检查

当编写函数库时，保证它们包括下面的代码行。

清单4-1 函数库框架

```
#pragma library;
#pragma option +l;
/* . . . */
#pragma endlibrary;
```

这促使编译器不去产生不被主模块引用的任何函数代码，并在一个清单文件里重新产生函数库代码。

4.5.3 模拟器环境

基于软件的模拟器作为一个测试环境，带来了巨大的灵活性。尽管不是物理的，但它能够被编写或者配置以精确地匹配特定的程序员模式和硬件特征。

当在一个现代PC上运行时，模拟速度不是一个问题：几百兆赫主频的PC能够轻松地模拟在1~10兆赫之间的通常MCU速度下的事件。

4.5.4 仿真器环境

采用仿真器显现出一种权衡：仿真器提供一个物理基础的测试环境，却不能重复产生特定的物理配置。它们只反映出它们实现的设计在某种程度上成功了。

仿真器主机软件接受某种调试文件格式。Byte Craft的.COD文件是一种这样的格式。它包括在执行数据里通常没有描述出的额外信息，例如，对应产生代码的每个段落的源代码行号。

根据这个额外信息，仿真器能够定位源代码或清单文件中的断点。您能够判断仿真器主机软件报告的寄存器值的上下文。

4.5.5 在测试套件里的目标系统

在原型硬件完成后，尽可能快地迁移候选软件到原型硬件是明智的。测试套件由一些简单的部件组成：开关、灯、小马达或其他简单的指示器。测试套件应复制测试部件所指定的工作环境的连接（和任何的反馈情况）。

对程序员来说，真正的挑战在于理解测试套件和现实世界的差异。令人高兴的是，您不必改变重要的常量，如预置数值。

结果

对于初始的代码审查，我们将使用C6805清单文件。无论在编码-编译循环期间，还是对其他人的工作进行代码审查时，这个清单文件包括很多有用的报告。

关于仿真器，我们将使用Motorola公司的MC68HC705JICS产品。仿真器使用一个串行电缆连接到一个PC机，并使用6805C8重新创建I/O端口并与主机系统通信。主机系统实际地评估j1a软件。仿真器是非实时的：例如，改变端口位的命令必须由PC传递到JICS板。

对于自动调温器，测试套件组成如下。

- 30V灯泡代表启动时的制热单元和制冷单元。
- 来自墙上插座的单元电源和地线。

第5章 嵌入式系统中使用C

考虑进预期的硬件环境，在进行精炼设计之后，就可以开始编码了。着手一个嵌入式工程的编码与编码一个桌面应用工程并没有多大差异。

最重要的是，目前惟一的软件环境是通过设备默认、全局声明、设置例程而建立的。main() 函数是真正的主函数。

表现嵌入式C开发特征的其他实践知识有：

- 内联汇编语言
- 设备知识
- 机械知识

5.1 内联汇编语言

尽管不是ANSI C所必需的，但大多数嵌入式开发编译器提供了在C程序里混入汇编语言的工具。完成这个功能的一个通用方法是使用预处理器指令。

Byte Craft编译器采用#asm和#endasm指令来声明汇编语言代码边界。在指令间的任何东西都将被已置入编译器的宏汇编器处理。

在C中使用的标号和变量也同样可以在被嵌入的汇编代码里得到。然而，编译器不会尝试优化这样的代码。编译器假定用户有一个好的理由来避免编译器的代码产生和优化。

微控制器的生产商将提供手工性的汇编语言程序设计的支持。有可能需要修改发生故障的操作代码以适应某个流水线，即编译器透明执行的某项事件。

wait()函数的下列两种定义显示了用C写的函数和用Motorola 68HC705C8汇编语言写的等价函数。

清单5-1 包含内联汇编语言的C函数

```
/* C 函数 */

void wait(int delay)
{
00EA
0300 B7 EA    STA    $EA
0302 3A EA    DEC    $EA    while(--delay);
0304 26 FC    BNE    $0302
0306 81      RTS
}
```

/*手工编写的汇编版本。注意：存储参数和函数返回值的代码还是要产生。几乎没有原因改变这样的事实：如果要避免使用一个局部变量，考虑声明参数为(BCL)registera或registerx，或其他等同的名字。*/

(续)

```

                                void wait2(int delay)
00EA                                {
0307 B7 EA      STA      $EA
                                #asm
                                LOOP:
0309 3A EA              DEC delay;
030B 26 FC              BNE LOOP;
                                #endasm
030D 81              RTS      }

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

5.2 设备知识

在嵌入式世界里，为一个控制器体系结构产生代码的编译器还要必须支持大量的些微差别的处理器族：具有不同RAM和ROM数量的芯片、更少或更多的端口、特殊的特征等。定制芯片的可能性也要考虑（例如，可编程屏蔽的ROM例程）。

标准C环境允许特定编译器扩展的定义中带有#pragma预处理器指令。该预处理器可在源代码里处理#pragma指令，或者可能由编译器处理这些指令。

#pragma指令常用于嵌入式开发中描述目标硬件的特殊资源，如可得到的内存、端口，以及特殊指令集。甚至处理器时钟速度也可以被指定，如果它对编译器来说很重要。下面各节描述Byte Craft编译器所必需的#pragma指令。

5.2.1 #pragma has

#pragma has描述了处理器特殊体系结构方面的特性。#pragma has指令限定符依赖于处理器族和编译器。

大部分#pragma has语句出现在设备头文件中。下列的例子显示了编译的代码在启用和禁止has MUL时的不同之处。

清单5-2 6805乘法（禁止#pragma has MUL指令）

```

                                void main(void)
                                {
00EB                                unsigned int result;
00EA                                unsigned int one;
00E9                                unsigned int two;

030E A6 17      LDA      #$17      one = 23;
0310 B7 EA      STA      $EA
0312 A6 04      LDA      #$04      two = 4;
0314 B7 E9      STA      $E9

```


(续)

```

0316 B6 EA    LDA    $EA        result = one * two;
0318 BE E9    LDX    $E9
031A CD 03 20 JSR    $0320
031D B7 EB    STA    $EB

031F 81      RTS
0320 B7 ED    STA    $ED        /* 乘法子例程 */
0322 A6 08    LDA    #$08
0324 B7 EC    STA    $EC
0326 4F      CLRA
0327 48      LSLA
0328 59      ROLX
0329 24 05    BCC    $0330
032B 8B ED    ADD    $ED
032D 24 01    BCC    $0330
032F 5C      INCX
0330 3A EC    DEC    $EC
0332 26 F3    BNE    $0327
0334 81      RTS

```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

清单5-3 6805乘法 (启用#pragma has MUL指令)

```

                                void main(void)
                                {
00EB                          unsigned int result;
00EA                          unsigned int one;
00E9                          unsigned int two;

03CE A6 17    LDA    #$17        one = 23;
0310 B7 EA    STA    $EA
0312 A6 04    LDA    #$04        two = 4;
0314 B7 E9    STA    $E9

0316 B6 EA    LDA    $EA        result = one * two;
0318 BE E9    LDX    $E9
031A 42      MUL
031B B7 EB    STA    $EB

031D 81      RTS

```

5.2.2 #pragma port

#pragma port指令描述在目标计算机上可用的端口。这个声明预留内存映射的端口位置，因而编译器不使用它们作为数据内存的分配。

#pragma port指令指示读或写或者两者的访问。某些条件下，I/O端口的电子设备可能禁止写到端口或从端口读。编译器能够报告不受欢迎的端口访问，如果它发现声明里有限制的话。除了对端口寄存器的保护，声明允许为端口提供一个益于记忆的名字，可以使用与这个端口相关联的名字来读或写它的输入或者输出状态。

下面定义了Motorola 68HC705C8里的两个端口以及与它们相关的数据方向寄存器。

清单5-4 用#pragma指令定义端口

```
#pragma portrw PORTA @ 0x0000;
#pragma portrw PORTB @ 0x0001;
#pragma portw DDRA @ 0x0004;
#pragma portw DDRB @ 0x0005;
```

编译器被告知有两个端口。名字PORTA指物理端口A的数据寄存器，它是可读和可写的，并且位于地址0x0000上。名字DDRA指物理端口A的数据方向寄存器，它是只可写的并且位于地址0x0004上。然后，采用C赋值语句就能够写值0xAA到端口了。

清单5-5 使用C赋值语句设置端口

```
DDRA=0xFF; /* 设置方向给输出 */
PORTA=0xAA; /* 设置输出引脚为10101010 */
```

特定芯片的资源最好通过使用#include来引入的头文件描述。ANSI C有一个关于#pragma指令的指定规则：如果一个#pragma指令不能被编译器识别，则编译器忽略它。这保证了不认识的#pragma指令将不会影响程序代码。

5.2.3 字节次序

程序员必须牢记于心的一条设备知识是处理器的字节次序（endianness），C并不直接处理字节次序，甚至在多字节移位操作时也如此。

在程序员想要直接操作多字节值的芯片的情况下，必须从生产商提供的信息中确定是高字节（大端）还是低字节（小端）作为第1个字节存于内存中。

由于微控制器的有限资源，一些古怪的事情出现了。COP8体系结构在内存中（针对间接操作）存储地址为高位优先，存储数据为低位优先。向栈中压入地址时与存储寄存器或RAM里的地址呈现不同的字节次序。

当建立符号表时，编译器通常采用最低（第1个）内存单元记录标识符的位置，不管处理器是何种字节次序。

5.3 机械知识

在嵌入式系统程序里用到的技术通常是基于特殊设备或外设操作的知识。现代操作系统应用程序接口（API）设计成向应用程序开发者隐藏这些技术。嵌入式C系统需要对外围设备进行第一手的控制，但还是提供了相当程度的泛化。

端口库所使用的一个有用技术是定义字母I和O作为端口控制寄存器的合适设置，而端口控制寄存器管理着数据方向。字母不能被单个地定义，它们被定义为8字母的序列，并且这个序列不可能在任何其他地方出现。

应用程序可能需要让某一个端口既作为输入又作为输出（例如，通过软件驱动一个双向并行端口），使用这些宏设置端口的数据方向提供了设备独立性。

清单5-6 设备独立的数据方向设置

```
#pragma portw DDR @ 0x05;

#include <port.h>
/* port.h包括如下所示的多个定义:

#define IIIIIIII 0b00000000
#define IIIIOOOO 0b00001111
#define OOOO0000 0b11111111

这里，输出设置DDR位为1('1')，
输入设置DDR位为零('0')。
它们能够重新定义为相反的设置
*/

/* ... 接下来 ... */

DDR = 00000000; /* 所有位设置为输出 */
DDR_WAIT();
/* ... 对端口执行写操作 ... */
DDR = IIIIIIII; /* 所有位设置为输入 */
DDR_WAIT();
/* ... 对端口执行读操作 ... */
```

低功耗操作能够通过重复地使处理器处于不活动模式而达到，直到某个事件产生中断后才停止。根据保护处理器寄存器和恢复时期的内容而预备的不同措施，处理器家族提供了STOP或WAIT操作的变种。C恰当地把这些变种表示为STOP（）或者WAIT（）宏。如果硬件停止不被支持，则该宏可能被重新定义以引发一个无限循环，跳转到重置向量，或者执行一个替代操作。

当按住一个键时，它“弹起”，这意味着几次快速按下和松开而不是仅仅一次才表明该键被

读入。为确保一次按键不被解释为几次弹起，必须包括去弹起的支持。当第一个键盘开关寄存到端口时，软件调用keypad_wait()函数以确立一个延时，然后，再次检查按键。如果按键不再处于被按下的状态，则这次按键被解释为一次弹起（或者一个错误），然后循环再次开始。当信号在延时前后都出现时，很可能是机械已经停止弹起操作并且按键被记录下来了。

5.4 函数库

函数库是模块化编译时代码重用的传统机制。嵌入式系统的C语言充分利用函数库这个组织工具。

- 跟通常一样，函数库是没有main()主程序的代码模块。
- 有关的头文件应该声明函数库里的变量和函数为extern类型。
- 链接过程要比桌面软件开发的链接过程更简单。这里没有必要归档目标文件，也不用考虑动态链接。
- 链接过程中，目标文件里仍然留有未被引用的函数，这在嵌入式软件里是不可接受的。在Byte Craft编译器里，#pragma library和#pragma endlibrary定界语句确认并不是在函数库里的所有例程都需要被链接进来。如果让编译器仅抽取被引用的例程，则节省的ROM空间是值得付出额外努力的。
- 检查为函数库产生的代码同检查主模块代码一样重要。在一个函数库里的#pragma option +1；语句将使编译器把源代码和汇编代码从函数库里加入到最终程序的清单文件中。

5.5 初看嵌入式C程序

传统上，程序员用C写的第一个程序是在计算机屏幕上显示“Hello World!”的消息。

在8位微控制器世界里，没有提供标准输入和输出的环境。一些C编译器提供了stdio库，但是输入和输出的解释是与具有管道和shell环境的桌面系统不同的。

下面是一个同“Hello World!”类似的人门程序。程序通过测试判断连接到一控制器端口的按键是否被压下。如果按键被压下，程序打开连接到该端口的指示灯，等待一会，然后关闭指示灯。

清单5-7 针对微控制器的“Hello World!”程序

```
#include <hc705c8.h>
/* #pragma portrw PORTA @ 0x0A; 在头文件里有这个声明
   #pragma portw DDRA @ 0x8A; 在头文件里有这个声明 */
#include <port.h>
#define ON 1
#define OFF 0
#define PUSHED 1

void wait(register a); /* wait 函数原型，略去函数代码 */

void main(void){
```

(续)

```
DDRA = 0b11111110; /* pin 0 为输出, pin 1为输入
                      其他无所谓 */
while (1){
    if (PORTA.1 == PUSHED){
        wait(1);          /* 是一个有效的接压吗 */
        if (PORTA.1 == PUSHED){
            PORTA.0 = ON;    /* 开灯 */
            wait(10);        /* 短延时 */
            PORTA.0 = OFF;   /* 关灯 */
        }
    }
}
} /* end main */
```



第6章 数据类型和变量

由于嵌入式控制器的受限环境，标准C变量和数据类型具有新的特征。

最大的改变发生在默认整数类型是8位或者16位。虽然从C的观点来看这是完全可以接受的，但是程序员以前常用的普通32位值需要调整到新的环境。默认情况下，Byte Craft编译器创建8位int，而long或者long int数据类型是2字节大小。

嵌入式编译器接受标准C类型，以及适合于嵌入式开发的几种附加数据类型。同时，嵌入式世界也给类型转换引入了一种新的观点。Casting是一种由编译器做起来更容易的任务，但是casting更容易失去信息，并且会干扰预定用于外设控制之类的上下文的值。

其他实质性的改变包括数据类型和变量，它们带来了重要的副作用。

- 常量或者已初始化变量将消耗更多的ROM和RAM。包括初始化的全局变量声明将在重置后立即自动产生放置一个值到已分配地址的机器代码。对于Byte Craft编译器，一个或者多个全局变量初始化在赋初始值给它们之前，产生代码以零初始化所有变量的内存。
- 类型为register的变量是可用的，但是在典型的8位体系结构下，寄存器的缺乏致使它们比通常情况更不稳定。
- 对于Byte Craft编译器，声明为在SPECIAL内存区域内的变量的一个简单求解或者赋值能够产生一个子例程调用。读或者该值的驱动器子例程如果正在与外围设备通信，则它要花大量的时间来执行。

除了内置类型，程序员能够像往常一样定义他们自己的数据类型。

当编译器遇到一个变量声明时，它检查到该变量以前没有声明过，就会分配一个大小合适的内存块。例如，char变量在默认情况下需要单字（8位）的RAM或者数据内存。数据类型修饰符影响变量分配内存的大小和方式。

存储修饰符影响内存何时被分配以及在被重用时怎样被释放。

- 一些变量在跨越几个模块时都仅被分配一次，甚至以前被编译的模块可能需要访问一个公共变量。编译单元，即函数库或者目标文件，必须使用extern存储类修饰符来表示它们是外部符号。
- 属于互斥范围的非静态变量可能被重叠在一起。嵌入式C对待变量范围同标准C一样，但是，在使用范围来帮助保存内存资源时需要付出额外的努力。
- 如果可能，在每个进入子例程的入口，编译器将重新初始化局部变量。这些变量实际上被声明为auto。声明为static的局部变量被单独放置在函数的开始处。如果它们有初始值，那么Byte Craft编译器按照全局变量初始化的方式仅给赋值一次。

关于每种C数据类型和存储修饰符的嵌入式的解释，见表6-1。

表6-1 数据类型修饰符和注释

修 饰 符	注 释
auto	对局部变量是不必要的。与static相对
const	在ROM里分配内存
extern	标志以后从一个库中对解析引用
far	依赖于目标的寻址机制
near	依赖于目标的寻址机制
signed	与unsigned相对，生成额外代码
static	在多个函数调用间保持局部变量
unsigned	在生成代码实现大的节省
volatile	(没有特别的注释；要得到更多信息，请咨询ISO标准)



6.1 标识符声明

嵌入式C编译器采用C声明来分配变量或者函数的内存。

在编译器读一个程序时，它在一个符号表里记录所有的标识符名称。编译器内部地使用符号表作为对跟踪标识符的引用；它们的名字、类型和它们代表的内存单元。大多数编译器支持至少31个字符长度的标识符。

有时直接指定变量的位置是必要的或者是程序员所期望的。Byte Craft编译器解释@操作符和紧跟在标识符后的数字为变量值被存储的位置。同时，在#pragma port语句里，@操作符也被用于端口寄存器和标识符之间的关联。这些标识符占有与RAM和ROM内存变量标识符相同的名字空间。

特殊数据类型和数据访问

RAM的每一个位都是精确的。尽管在外围设备上没有使用到的RAM没有出现在处理器当前的地址空间里，但精细的技术能够使它出现。声明一个内存空间为SPECIAL需要程序员写例程，以读和写来自外设和向外设写的数据。此时，还要权衡性能问题。

清单6-1 SPECIAL：内存、驱动方法和变量声明

```
#pragma memory SPECIAL eeprom [128] @ 0x80;
#define eeprom_r(LOC) I2C_read(LOC)
#define eeprom_w(LOC,VAL) I2C_write(LOC,VAL)

int eeprom i;
```

访问声明为特殊内存区域里的变量将花费一些时间，但是编译器可以作出透明的处理。

6.2 函数数据类型

函数数据类型决定一个子例程能够返回的值。例如，一个int函数类型返回一个有符号的整

型值。

如果没有指定返回类型，则任何函数返回一个int。尽管返回值是不能预料的，但嵌入式C编译器甚至在main()函数里也提供了这种返回。为了避免混淆，程序员应该总是声明main()函数的返回类型为void。

一些其他的特殊命名函数有预定义的类型。例如，实现中断编码的函数有void类型，除非有某种方法让中断返回一个值。Scenix SX（一种16位芯片）为支持虚拟外设而返回一个值，因此，它的中断处理程序将有int函数数据类型。

参数数据类型指示传递给函数的值以及为存储它们应保留的内存。一个没有任何参数声明的函数（例如，带有空圆括号的函数）被认为没有参数，准确地声明应该为(void)。

编译器依赖于目标芯片分配内存时差别很大。例如，Byte Craft编译器传递前两个（字节大小）的参数是通过处理器的一个累加器和另一个寄存器进行的。如果局部内存被特别声明，编译器将在那个空间之外的位置分配参数。

6.3 字符数据类型

C字符数据类型char存储字符值并且被分配一个字节的内存空间。字母信息最常见的使用是输出到一个LCD面板或者从一个键盘设备输入，这里使用的每个字母由一个字符值指示。

6.4 整数数据类型

整数值能够被存储为int、short或者long数据类型。int值的大小在8位体系结构中通常是16比特。Byte Craft编译器的默认int大小是在8位和16位间可切换的。

数据类型short用于补偿int字长的变化。在许多传统C平台上，int类型的字长多于2个字节。在int字长大于2字节的平台上，short类型应该是2字节长。在int类型的字长是1或者2字节的平台上，即大多数8位微控制器，short数据类型典型地占用一个单字节。

如果程序需要操作大于一个int数据的值，则程序员使用long数据类型。在大多数平台上，long数据类型占据int数据类型2倍的内存空间。对于8位微控制器，long数据类型通常占用16位。

需要特别注意的是，long整数值几乎总是存储在大于计算机自身数据总线大小的内存块里。这意味着，当一个程序使用long值时，编译器通常必须产生更多的机器指令。

long和short是有用的，因为它们不大可能在具有通常的8位数据类型的目标和另一个16位值的目标平台间改变。假设有一个可切换的int类型，程序员可以通过在需要8位值时使用short，在需要16位值时使用long来维持代码的可移植性。

像int一样，short和long数据类型默认使用一个符号位，因而它们能够包含负数。

指明长度的整数

Byte Craft编译器识别int8、int16、int24和int32数据类型，它们是含有相应数目的位的整数。这些整数数据类型消除了变化的或者可切换的整数长度的二义性。

6.5 位数据类型

嵌入式系统需要高效地处理比特大小的值。

ISO/IEC 9899:1999详细描述了_Bool类型。_Bool类型的变量可能是0或者1,这是对C标准的一个新的补充。

Byte Craft编译器针对位大小的量级提供了两种类型:bit和bits。一个bit值是一个单独的位,编译器依赖于处理器的能力存放和管理它。

一个bits变量是一个8比特的结构,被放在一起管理并且可以使用结构成员符号单个寻址。能够指派一个字节值直接给一个bits变量,然后对单个的位寻址。

清单6-2是一个针对MC68705J1A的例子。

清单6-2 比特大小的变量类型

		bits switch_fixup(void)
		{
00EB 0000		bit heat_flag;
00EB 0001		bit cool_flag;
00EA		bits switches;
0300 00 01 04	BRSET 0,\$01,\$0307	heat_flag = PORTB.0;
0303 11 EB	BCLR 0,\$EB	
0305 20 02	BRA \$0309	
0307 10 EB	BSET 0,\$EB	
0309 02 01 04	BRSET 1,\$01,\$0310	cool_flag = PORTB.1;
030C 13 EB	BCLR 1,\$EB	
030E 20 02	BRA \$0312	
0310 12 EB	BSET 1,\$EB	
0312 B6 01	LDA \$01	switches = PORTB;
0314 B7 EA	STA \$EA	
0316 0B EA 05	BRCLR 5,\$EA,\$031E	if(switches.5 && heat_flag) switches.1 = 0;
0319 01 EB 02	BRCLR 0,\$EB,\$031E	
031C 13 EA	BCLR 1,\$EA	
		return(switches);
031E 81	RTS	
		}

6.6 实数

当许多桌面计算机应用广泛利用实数或者浮点数(小数点两边都有数字的数)时,8位微控

制器应用并不如此。存储和操作浮点数需要大量的资源,这将使8位计算机难以负担。通常,由此获得的好处不值得所消耗的资源。

在C里代表实数的基本数据类型是float类型。目标计算机的最大实数值在一个名为values.h的头文件中定义，其符号常量为MAXFLOAT。

C编译器一般为float变量分配4个字节，提供的精度大约是小数点右边6位数字。也能够用double或者long double数据类型获得更高的精度。典型地，编译器为double变量分配8个字节，为long double变量分配更多字节。double值的精度大约是15位小数，long double可能精确到更多的小数位。

IEEE 754又是一种格式，针对浮点数规定了4字节或者3字节格式。

我们可以指派一个整数值给一个浮点数类型，但是必须包括一个小数点和小数点后面的一个0。

```
myFloatVariable = 2.0;
```

6.7 复杂数据类型

复杂数据类型包括指针、数组、枚举类型、联合和结构。即使在资源受限的8位微控制器里组织嵌入式程序，复杂数据类型也是很有用的。

6.7.1 指针

指针变量的实现严重地依赖目标处理器的指令集。如果处理器有一种间接或者变址的寻址模式，则产生的代码将会更简单一些。

要着重注意的是，哈佛体系结构有两种不同的地址空间，因此指针的解释可能不同。RAM位置的解引用与ROM位置的解引用使用不同的指令。

同样需要注意的是near指针和far指针的差异，这在代码产生方面的差异是显著的。更多的信息请参见6.9.4节。

6.7.2 数组

当声明一个数组时，必须同时声明数组的类型和它包括的元素个数。例如，下面声明了一个包含了8个int元素的数组：

```
int myIntArray[8];
```

在声明一个数组后，预留一个单一连续的内存块用于保存它。这就是必须指定数组大小或者在声明时指派内容的原因。

清单6-3 已初始化和未初始化数组

[illegible]

(续)

```

/* 下面已初始化 */
0312 01 08 02 07 03 06 04 05    const int myconsts[] =
                                {1,8,2,7,3,6,4,5};

/* 常量数组没有代码产生 */

/* ...为清晰起见,省略main()代码 ... */

/* 初始化代码。第1段清除所有的变量内存。第2段初始化my2array。最后,跳
转到main()。 */

07FE 03 32
0332 AE C0    LDX    #$C0
0334 7F      CLR    ,X
0335 5C      INCX
0336 A3 EB    CPX    #$EB
0338 26 FA    BNE    $0334
033A 5F      CLRX
033B D6 03 48 LDA    $0348,X
033E E7 C8    STA    $C8,X
0340 5C      INCX
0341 A3 08    CPX    #$08
0343 26 F6    BNE    $033B

0345 CC 03 1A JMP    $031A

0348 01 02 04 08 10 20 40 80

```

在嵌入式C程序设计时使用数组有某些限制或者不利之处,这是因为在数组中采用变址的可用方法有限。

Byte Craft编译器禁止struct和union数组。这种限制是因为寻址该数据结构的成员的难度太大,而成员本身要作为数组成员来寻址。为克服这种限制,程序员采用基本数据类型的几个全局数组,并通过上下文将它们组织在一起。

6.7.3 枚举类型

枚举类型是已命名的数值的有限集合。

对枚举元素的任何清单,编译器默认提供从0开始的整数值范围。虽然在许多情况下用这种方法标识集合里的元素是足够的,但在嵌入式C里,可能希望关联枚举集合到一个由设备决定的级数。枚举元素能够被设置成下面两种方式的任何整数值。

(1) 为每一个枚举元素指定值。下面的例子来源于COP8SAA7 WATCHDOG服务寄存器WDSVR。该寄存器的位6和7选择服务窗口的上限，而服务窗口选择WATCHDOG服务时间。

清单6-4 为枚举元素指定整数值

```
enum WDSel { Bit7 = 7 , Bit6 = 6 } ;
```

既然字符常量被存储为整数值，那么它们能够被指定为枚举清单的值。

```
enum DIGITS {one='1' , two='2' , three='3' } ;
```

将存储相应于元素清单中指定的每个数字的机器字符集（通常为ASCII）的整数值。

(2) 为一个或者多个枚举元素指定一个起始值。默认情况下，编译器指派值0给清单的第一个元素。可以设置清单从其他值开始。

清单6-5 为枚举元素指定一个起始值

```
enum ORDINALS {first = 1, second, third, fourth, fifth} ;
```

当编译器遇到在枚举清单里没有指派值的元素时，它从已经被指定了值的最后一个值开始计数。例如，下面的枚举清单为其元素指定了适当的值。

清单6-6 给一个枚举清单指派整数值

```
enum ORDINALS {first = 1, second, fifth = 5, sixth, seventh} ;
```

6.7.4 结构

结构支持程序数据的有意义的组织。建立可理解的数据结构是提高新程序效率的一个关键。

下面的声明创建一个扩展时间计数器的结构化类型，并在该结构里描述每一个元素。结构display被定义为具有如下组件：hours, minutes, seconds和一个AM/PM标志。接下来，变量timetext被声明为具有struct display类型。

清单6-7 声明一个结构的模板

```
struct display {
    unsigned int hours;
    unsigned int minutes;
    unsigned int seconds;
    char AorP;
};
```

```
struct display timetext;
```

Byte Craft编译器允许使用位域，即单个域少于8比特的结构。使用位域可以使声明的结构只占用必不可少的空间：几个域可能占用单一字节。

下面的例子是针对Motorola MC68HC705C8的，它把定时器控制寄存器（TCR）位定义为称作TCR的结构中的位域，并使用该结构配置定时器输出比较。



清单6-8 结构中的位域

```
struct reg_tag {
    int ICIE : 1; /* 域ICIE, 1位长 */
    int OCIE : 1; /* 域OCIE, 1位长 */
    int notUsed : 3 = 0; /* 域notUsed是3位并被设置为0 */
    int IEDG : 1; /* 域IEDG, 1位长 */
    int OLVL : 1; /* 域OLVL, 1位长 */
} TCR;

/* 配置定时器: */

TCR.OLVL = 1; /* TCMP引脚在输出比较成功时变为高电平 */
```

Byte Craft编译器允许一个比特域横跨两个字节。然而，并不是所有的编译器支持这种优化。在最坏的情况下，下面的结构可能放置整个第2个域到与第1个域相隔离的内存空间。

清单6-9 依赖位域存储的编译器

```
struct {
    unsigned int shortElement : 1; /* 1位长 */
    unsigned int longElement : 7; /* 7位长 */
} myBitFields; /* 可能是1字节，最坏情况是2字节 */
```

编译器存储结构元素到结构位域的顺序因编译器的不同而不同。

位域元素的运算与相同大小的无符号int类型一样。因此，占用一个单比特的元素可能取整数值1或者0，而占用2位的元素可能取从0~3之间的任何整数值。还能够像对待int类型一样，在计算和表达式里使用每个域。

6.7.5 联合

开发传统平台的C程序员不经常使用union数据类型，但它是对嵌入式系统开发者非常有用的资源。union类型基于几个相关数据类型中的一个解释存储在单一内存块的数据。

嵌入式系统中的union类型的一个常见用法是创建一个可擦写缓冲区变量，由它保存不同类型的数据。在需要一个临时变量的每个函数里，通过重用一個16位内存块可节省内存。下面的例子显示了创建这样一个变量的声明。

清单6-10 使用联合类型创建一个可擦写缓冲区

```
struct lohi_tag{
```

(续)

```

        short lowByte;
        short hiByte;
    };
    union tagName {
        int asInt;
        char asChar;
        short asShort;
        long asLong;
        int near * asNPtr;
        int far * asFPtr;
        struct hilo_tag asWord;
    } scratchPad;

```



针对联合的另一个常见用途是帮助把数据作为不同类型来访问。例如，Microchip PIC16C74有一个名为TMR1的16位定时器/计数器寄存器。TMR1是由名为TMR1H（高字节）和TMR1L（低字节）的两个8比特寄存器组成的。

不求助于指针操作而随意访问8位寄存器中的任何一个是令人满意的，union将满足这种数据访问。

清单6-11 使用联合类型访问不同类型的数据

```

struct asByte {
    int TMR1H; /* 高字节 */
    int TMR1L; /* 低字节 */
}
union TMR1_tag {
    long TMR1_word; /* 作为16位寄存器访问 */
    struct asByte halves;
} TMR1;

/* ... */

seed = TMR1.halves.TMR1L;

```

既然编译器为整个联合使用一个单一内存块，那么，它就必须为联合中的最大元素分配足够大的内存块。编译器将在内存块最低地址处对齐每个元素最前面的位。如果指派一个16位值给scratchPad，然后按8位值读它，编译器将返回存储数据的第1个8比特。

如果任意抽取16位变量的一个字节，返回的值将依处理器体系结构字节次序的不同而不同。正如5.2.3节所提到的，C并不直接处理字节次序。

6.8 typedef

关键词typedef是指依照现存类型定义一个新的变量类型。编译器主要关心新类型的大小，

以便确定要预留的RAM或者ROM的数量。

清单6-12 用typedef定义新类型

```
typedef int new_int;
new_int result; /* 在不同的上下文里表示相同的值域 */

typedef struct {
    char * name;
    int start;
    int min_temp;
    int max_temp;
} time_record;

time_record targets[] {
    { "Night", 0, 20, 25},
    { "Day", 5*3600, 20, 25},
    { "Evening", 18*3600, 20, 25},
}
```

超星阅读器提醒您：
使用本复制品
请尊重相关知识产权！

6.9 数据类型修饰符

C语言允许修改简单数据类型的默认特征。这些数据类型修饰符主要改变允许的值的范围。

类型修饰符只应用于数据，而不针对函数。程序员可以在变量、参数和函数的返回值里使用它们。

某些类型修饰符能够同任何变量一起使用，而其他则只用于特殊类型的一个集合。

6.9.1 数值常量修饰符：const和volatile

编译器优化程序的能力依赖于多个因素，其中之一是程序里数据对象的相对持久性。默认时，程序里使用的变量是根据开发者给出的指令改变数值的。

有时，程序员想创建不能改变数值的变量。例如，如果代码中用到了 π ，即常数PI，则应该在一个常数变量里指定该数值的一个近似值。

```
const float PI = 3.1415926 ;
```

当程序被编译后，编译器为PI变量分配ROM空间并且不允许在代码中改变它的数值。例如，下面的赋值将在编译时产生一个错误。

```
PI=3.0 ;
```

在嵌入式C里，常量数据值的存储是从计算机程序的内存空间分配的，通常是ROM或者其他的非易失性存储体。

对于Byte Craft编译器，下面这样的声明


```
const int maximumTemperature = 30 ;
```

声明了一字节常量，其初始值为十进制数30。如果任何特殊的技术被要求用来装载数据到一个寄存器，则编译器对常量将保留远远多于一个或两个字节的空間。由于受到体系结构的限制，某些平台需要常量成为嵌入在ROM子例程中的一个多字节装载语句的参数；为访问常量值，处理器要执行专用的装载语句。

Volatile变量是超出正执行软件范围其值就可能改变的变量。例如，一个“存储”在端口数据寄存器位置的变量将随着端口值的改变而改变。

使用volatile关键词通知编译器，它不能依赖于变量的值且不基于被赋予的值执行任何优化。

6.9.2 允许值修饰符：signed和unsigned

默认情况下，整数数据类型能够容纳负数值。可以限制整数数据类型只取正数值。整数数据类型的符号值是用signed（有符号）和unsigned（无符号）关键词来指定的。

signed关键词迫使编译器使用整数值的高位作为符号位。如果符号位被设置为1，则变量的其余部分被解释为一个负数值。short，int和long数据类型默认为是有符号的，char数据类型默认为是无符号的。为了创建一个有符号的char变量，必须使用下面的声明：

```
signed char mySignedChar ;
```

如果单独使用signed和unsigned关键词，编译器则假定正在声明一个整数值。因为int值是默认有符号的，所以程序员很少使用语法signed mySignedInt ；。

6.9.3 大小修饰符：short和long

修饰符short和long指示编译器应为一个int变量分配多少空间。

short关键词修改int，使它同—个char变量具有相同大小的位（通常是8位）。

```
short int myShortInt ;
```

如果单独使用short关键词，编译器会认为该变量是一个short int类型。

```
short myShortInt ;
```

long关键词修改int，使它同—个正常的int变量一样长。

```
long int myLongInt ;
```

同样，省略long声明后的int仍被认为是long int。

6.9.4 指针范围修饰符：near和far

关键词near和far深受目标计算机体系结构的影响。

near关键词创建一个指向可寻址内存低端部分的目标的指针。这些指针占用内存的单一字节，并且它们能够指向的内存单元被限制到256个位置，通常是0x0000 ~ 0x00ff的范围里。

```
int near * myNIntptr ;
```

far关键词创建一个能够指向内存中任何数据的指针：

```
const char * myString = "Constant String" ;
```

```
char far * myIndex = &myString ;
```

这些指针需要两字节的内存，允许它们占据0x0000 ~ 0xffff之间的任何合法地址。far指针通常指向用户ROM里的目标，如用户定义的函数和常量。



6.10 存储类修饰符

存储类修饰符控制已声明的修饰符的内存分配。C支持4种存储类修饰符，它们可以用于变量声明：extern, static, register和auto。只有extern用于函数声明。

ISO标准规定typedef作为第5个修饰符，但它解释这只是为了方便。typedef已经在6.8节描述。

当编译器读一个程序时，它必须确定如何为每一个标识符分配存储空间。完成这个任务的过程被称作链接。C支持3类链接：外部链接、内部链接和无链接。C使用标识链接以分类对相同标识的多个引用。

6.10.1 外部链接

带有外部链接的标识引用在整个程序里调用内存中相同的目标。带有外部链接的标识的定义必须是惟一的；否则，编译器将给出一个重复符号定义的错误。默认情况下，程序中的每个函数都有外部链接。同样在默认情况下，全局范围的任何变量也都具有外部链接。

6.10.2 内部链接

在每个编译单元里，带有内部链接标识的所有引用指向内存中的相同目标。这就意味着，在程序的每个编译单元里，带有内部链接的标识的定义只能是惟一的。因为采用#include指令，一个编译单元能够多于一个文件。

默认情况下，C里没有目标具有内部链接。具有全局范围（定义在任何语句块外）并且具有static存储类修饰符的任何标识拥有内部链接。同样地，具有局部范围（定义在语句块里）并且具有static存储类修饰符的任何标识拥有内部链接。

尽管能够创建具有内部链接的局部变量，划分范围的规则把局部变量的可见性限制到它们的封闭语句块里。这意味着能够创建这样的局部变量，允许它们的值存在于它们所出现的语句块的即时生存期之外。正常情况下，计算机在多个不同的语句块间共享局部变量空间。如果一个局部变量被声明为static，则仅为该变量分配一次内存空间：发生在遇到变量的第一时间。

注意 不像其他的内部链接目标，静态局部变量在编译单元里不必是独一无二的。它们必须在包含它们范围的语句块里是独一无二的。

内部链接的目标通常远远不如外部链接或者无链接的目标出现得频繁。

6.10.3 无链接

无链接标识的引用在语句块里指向内存中相同的目标。如果在一个语句块里定义了一个变量，则只能提供这种定义一次。

默认情况下，在一个语句块里声明的任何变量都是无链接的，除非static或者extern关

关键词被包含在声明中。

6.10.4 extern修饰符

假设库函数`int Calculate_Sum()`在一个库源文件里已经声明了。只要以前声明过像这样具有外部链接的标识,它就能在同一个编译单元里的任何地方使用。

如果要在任何别的编译单元里使用这个函数,必须告诉编译器,函数的定义已经或可以得到。这个概念同原型化一个函数是等同的,除了实际的定义不会出现在相同的编译单元。函数定义位于编译单元的外部。

为声明一个外部函数,使用`extern`关键词。

```
extern int Calculate_Sum( );
```

当编译器遇到一个外部函数声明时,就将它解释为有函数名、类型和参数的一个函数原型。关键词`extern`宣告函数定义位于另一个编译单元。编译器把解析这个引用的任务推给链接器。

如果建立了一个应用于许多程序的函数库,并创建了一个包含`extern`函数声明的头文件。那么,在编译单元里包括这个头文件,以便程序员自己的代码可以得到函数库里的函数。

如同函数一样,全局变量也有外部链接。全局变量是一个展现函数库的通用配置设置的好方法。这样可以避免额外的函数调用。

为建立一个在编译单元外部可读或可设置的全局变量,必须在它的源文件里正常地声明它,并在一个头文件里用`extern`声明它。

```
extern int myGlobalInt ;
```

编译器解释一个外部声明为一个通知:实际的RAM或ROM分配发生在另一个编译单元里。

6.10.5 static修饰符

默认情况下,在全局空间里声明的所有函数和变量具有外部链接并且对整个程序是可见的。有时需要具有内部链接的全局变量或者函数:它们应该在一个单一的编译单元里是可见的,但在外部则不可见。使用`static`关键词限制变量的范围。

清单6-13 使用静态数据修饰符限制变量的范围

```
static int myGlobalInt ;
static int staticFunc(void) ;
```

这些声明创建不可被其他编译单元访问的全局标识。

关键词`static`的使用几乎与局部变量相反。它在其被声明的块里创建该块的一个永久性变量。例如,对于跟踪一个递归函数调用它自身的次数(即函数的深度)这样一个不常见的任务,可以使用一个静态变量完成。

清单6-14 使用静态变量跟踪函数深度

```
void myRecurseFunc(void) {
    static int depthCount=1;
```

(续)

```

    depthCount += 1;
    if ( (depthCount < 10) && (!DONE) ) {
        myRecurseFunc();
    }
}

```

myRecurseFunc包含一个if语句，当递归太深时退出递归。静态变量depthCount用于跟踪当前的深度。

正常情况下，当函数被调用的时候，计算机重新初始化它的自动局部变量（或者至少使它们处于可疑状态）。然而，静态变量的内存只被初始化一次。静态变量depthCount在函数调用期间保持它的值。

因为depthCount在myRecurseFunc()语句块内部定义，所以相对这个函数之外的任何代码都是不可见的。

6.10.6 register修饰符

当用register修饰符声明一个变量时，通知编译器优化访问该变量的速度。典型地，C程序员在声明一个循环计数变量时使用这个修饰符。

清单6-15 使用寄存器数据类型修饰符

```

{
    register int myCounter = 1;
    while (myCounter<10) {
        /* ... */
        myCounter += 1;
    } /* end while */
} /* 封闭块迫使重新分配（收回）myCounter的内存 */

```

不像其他存储类修饰符，register只是给编译器一个简单的建议。如果没有用于分配的寄存器，则编译器可能使用正常内存空间。

因为在8位机器上寄存器的匮乏以及空间优化胜过速度，所以对嵌入式程序员而言，register关键词没有太大的用处。

注意，在上面例子里使用的技术做了两件事：它在一个语句块里放置了限于该语句块的register声明，和在该语句块内的while循环。这样最小化了使一寄存器专用于一特定变量的成本。它同时强迫编译器在循环结束后立即释放myCounter的存储：如果编译器使用一个寄存器存储myCounter，那么它占用寄存器的时间不会长于必需的时间。

6.10.7 auto修饰符

关键词auto指示一个临时变量（与static相反）。C在一个语句块内部不支持函数，所以

auto只能用于局部变量。既然默认情况下在一语句块里声明的所有变量是无链接的，那么使用auto关键词的惟一理由是提高代码清晰度。

清单6-16 使用auto数据修饰符

```
int someFunc(NODEPTR myNodePtr) {  
    extern NODEPTR TheStructureRoot;  
    /* 指向数据结构root的全局指针 */  
    auto NODEPTR tempNodePtr;  
    /* 为结构操作而声明的临时指针 */  
    /* ... */  
}
```



在这个例子里，声明tempNodePtr为auto变量以便清晰地表明不同于全局指针TheStructureRoot，tempNodePtr只是一个临时变量。

第7章 C 语言的语句、结构及操作

使用C语言来设计程序的好处之一就是它可以用数学表达式。汇编语言除了只能进行简单的常量运算之外，还迫使你采用严格的过程结构。C语言提供赋值语句、逻辑和数学表达式，以及控制结构，从而可以使用通用数学符号和有用的隐喻来表述意图。

使用本文档需
请尊重相关知识产权

7.1 块中的联合语句

可以为自己的函数建立语句块，以及在其他时间为控制语句体建立语句块。例如，while语句的一般格式如下：

```
while (condition) statement;
```

因为可以用语句块来代替任何一条单一的语句，while语句在大多数情况下会表示如下：

```
while (condition) {  
    statements  
}
```

7.2 函数

当编译器遇到函数定义时，它生成机器指令来完成这一功能，并预留足够的程序内存来存储函数中的语句。这一函数的地址可以在符号表中找到。

函数定义包括一个含有全部函数语句的语句块。即使一个函数中只有一条可执行语句，它也必需封装在一个语句块中。

嵌入式C语言支持函数原型。函数原型声明确保编译器了解该函数及它的参数类型，尽管它的定义在编译器的输入中已有。函数原型协助检测前向调用。函数名作为标识符记录下来，因此在其定义之前就可将其包括在代码中。

函数原型的头文件为函数库文件的使用提供了基础。

在C语言中，函数调用的语法是函数名后跟在圆括号中的实际参数表。

函数调用是嵌入式C语言与传统的C语言的显著区别之一。允许的参数数量和参数传送方式明显不同。

产生额外副作用的函数更难以维护和调试，特别是对那些开发队伍的成员来说。如果想要安全地使用这些抽象的函数，仅需知道那些输入和输出的数据——函数接口。当函数产生了额外的副作用时，就需要知道它的接口及其特性，以便安全地操作它。

一些C语言的编程人员坚持认为那些产生副作用的函数应返回一个值，以指示其执行是成功、失败还是出现错误。由于只读存储器ROM空间十分宝贵，使得评估函数返回状态所需要的代码变得很奢侈。

函数参数

对于嵌入式处理器，C语言在函数调用上设置了一些独特的限制。一些编译器限制传递到函数中的参数数量。两个字节的参数（或一个16位的参数）可以在一般处理器的寄存器（累加器和变址寄存器）中传送。

为了通过引用传递参数，像往常一样，要传递一个指针。在6.7.1节中查阅有关使用指针的相关代价的额外信息。

无参数的函数可以表示为一个空的参数清单。

```
int myFunc{ }
```

但最好还是养成用void参数型来表示这些没有参数的函数的良好习惯。

```
int myFunc(void)
```

在嵌入式程序中，main()不接受任何参数。

7.3 控制结构

虽然一些嵌入式C语言程序的流程在最初出现时会让人有一种奇特的感觉（例如while(1)就很典型），但它们与个人计算机中的C语言程序并没有什么本质的区别。

7.3.1 main()函数

操作系统不调用的嵌入式程序却有一个传统的main()函数和一个明确的返回值规范，这看起来有一点不合情理。那又是什么调用了main()呢？这一函数又将返回到哪里呢？

嵌入式C语言保留了main()功能，以便与标准C语言兼容。main()的返回类型一般应显式地声明为void。正如6.2节中提到的那样若省略声明其类型，就会让人理解为一个int的返回。

这样，main()函数可以执行其他函数的代码和接收返回的值。要记住，如果必要的话，你的被调用函数要经过原型化才能被main()调用。

7.3.2 初始化函数

嵌入式C语言程序允许专门的初始化例程。_STARTUP()就是这样一种Byte Craft编程器理解的函数之一。如果使用它，它的语句将会在控制转移到main()之前执行。

可以使用一套独立的初始化函数来更好地组织初始化任务。设备相关的硬件初始化必须针对每一个目标设备的特点而重写，且能存在于_STARTUP例程中或等价函数中。

7.3.3 控制语句

嵌入式系统开发人员经常使用程序控制语句，而其他程序员却尽量避免这样做。例如，C语言中的goto语句和汇编语言中使用的显式跳转指令或无条件转移指令有相同的上下文。

7.4 选择结构

C语言为程序员提供三种不同的选择结构。选择结构检测表达式以确定执行哪一条语句或哪

超星阅读器提醒：
使用本复制品
请尊重相关知识产权！

一个语句块。

如期望的那样，if..else是可用的。C条件操作符同样有效。

```
If (expression) statement else statement
result = expr? result_if_true : result_if_false
```

switch..case结构是在几种不同的可能执行的代码路径中进行选择的结构。switch..case结构被编译成类似if..else串的结构。

清单7-1 switch和case

00EB			int choice;
			switch(choice) {
			case 1: return 5;
0304 A1 01	CMP	#\$01	
0306 26 03	BNE	\$030B	
0308 A6 05	LDA	#\$05	
030A 81	RTS		
030B A1 02	CMP	#\$02	case 2: return 11;
030D 26 03	BNE	\$0312	
030F A6 0B	LDA	#\$0B	
0311 81	RTS		
0312 A1 03	CMP	#\$03	case 3: return 37;
0314 26 03	BNE	\$0319	
0316 A6 25	LDA	#\$25	
0318 81	RTS		
			default: return 9;
0319 A6 09	LDA	#\$09	
031B 81	RTS		

Byte Craft编译器扩展了case标号以处理一般的编程问题。如果编译器对于每一个case标号只能接受一个整数值，以下两个例子就需要编写更多的代码。

清单7-2 Byte Craft编译器对case的扩展

```
case '0'..'9': /* 接收数值的范围从 '0' 到 '9' */
case 0x02,0x04: /*接收可选择的值 */
```

这种结构的优点是避免了在一些案例中对每一个整数值重复比较switch变量。这样，编译器就可以生成一些简单的比较方式来处理一定范围的值或可选值列表中的值。

清单7-3 包含一定范围数值的一个case

```
case '0'..'9':
{
```

```

0473 A1 30    CMP    #$30
0475 25 24    BCS    $049B    /* 小于时转移 */
0477 A1 3A    CMP    #$3A
0479 24 20    BCC    $049B    /* 大于时转移 */
047B AE CA    LDX    #$DA
047D CD C5 4B JSR    $054B    scanf(&temperature,ch);

```

7.5 循环结构

C语言的控制结构允许在编码执行的路径中做出选择，同时也提供一个循环结构来控制程序流。循环结构允许重复一个语句集。

在嵌入式C语言中，while扮演着一个有趣的角色。我们常有意地使用while来构造一些无限循环。嵌入式控制器通常“无限”地执行一个单独程序，所以这种结构是大有用处的。

另外一种办法，即使用goto，要求使用一个标号。总之，程序编译器会用一个无条件跳跃或分支语句来执行while(1)选择语句。

清单7-4 一个无限循环的程序框架

```

                                void main(void)
                                {

                                while(1)
                                {
0300 B6 01    LDA     $01          PORTB = PORTB << 1;
0302 48       LSLA
0303 B7 01    STA     $01
0305 20 F9    BRA     $0300      }
                                }

```

7.5.1 控制表达式

任何一个循环结构的关键组件都在于这个控制表达式。在每一次循环的某一处都要检测控制表达式。如果控制表达式的值为0，那么程序将执行这一循环结构后的第一个语句。如果表达式的值为1，那么程序将继续执行循环结构中的语句块。

7.5.2 break和continue

C语言系统提供了两种方式来退出循环结构：break和continue语句。当在循环结构中遇到这两个语句中的任何一个时，循环结构中的剩余语句被忽略。

使用break语句，系统就会立刻从循环结构中退出。当一个break语句在循环结构中出现时，循环会立刻中止执行，系统转而执行循环结构之后的语句。

如果想直接跳到下一次循环中，而不完全退出整个循环时，可以使用continue语句。当一个continue语句在循环结构中出现时，程序会立刻进行到循环结构的末尾。

如果continue语句与while或for循环语句一起使用，那么，执行过程会从本语句块的末尾直接跳到循环顶端的控制表达式上。如果与do循环语句一起使用，执行过程会从本语句块的末尾直接转到循环底端的控制表达式上。在所有的情况下，其结果都是一样的，即continue语句不会绕过循环控制表达式，但它会跳过此次循环中尚未执行的任何语句。

对于break语句，最常见的使用方式就是在switch..case结构中。因为switch..case不是一个循环结构，switch..case中的continue语句转到包含此结构的循环结构（如果有的话）。

清单7-5 在loop和switch语句中的break和continue语句

00EB				char ch;
				while(1)
				{
030D AD F1	BSR	\$0300		ch = getch();
030F B7 EB	STA	\$EB		
				switch(ch)
				{
				case '0'..'9':
				{
0311 A1 30	CMP	#\$30		putch(ch);
0313 25 08	BCS	\$0310		
0315 A1 3A	CMP	#\$3A		
0317 24 04	BCC	\$0310		
0319 AD E8	BSR	\$0303		
031B 20 10	BRA	\$0320		break; /* 跳到switch之后 */
				}
				case 'A'..'C':
				{
031D A1 41	CMP	#\$41		continue; /* A~C的执行被忽略 */
031F 25 04	BCS	\$0325		
0321 A1 44	CMP	#\$44		
0323 25 E8	BCS	\$0300		/* 返回顶端，即到getch()语句之前 */
				}
				case 'D':
				{
0325 A1 44	CMP	#\$44		LCD_send_control(LCDC_R);
0327 26 04	BNE	\$0320		


```

0329 A6 05    LDA    #$05
032B AD 08    BSR     $0308
                /* 失败! */                break;
                }
            }

/* while(1)循环中的其他语句出现在这里 */

03FF 20 DE    BRA     $030D

```

7.6 操作符和表达式

使用C语言来进行嵌入式编程可以减轻手工编写大量算术运算所带来的枯燥乏味。通用微处理器的32位整数除法运算可以由一条指令完成，而8位控制器除了需要简化的数学运算外，还需要一系列装入和存储操作来执行同样的工作。

在嵌入式系统中，有一个不断被强调的重点就是关于按位操作。不管是考虑外设操作还是存储器效率，编译器都会尽可能地去使用位操作指令来实现按位运算符。

7.6.1 标准数学操作符

乘法类指令有时在硬件中是可以使用的。如果某一指令是用于改善体系结构的，编译器则需要配置，以生成使用该指令的代码。Byte Craft编译器可以利用一种可选择的乘法指令，该指令带有在设备头文件中的适当的#pragma has指令。有关#pragma has指令的更多信息请参见5.2.1节。

如果没有此类指令可以利用，编译器将提供乘法以及除法和模运算作为函数。如果需要使用这些运算的话，编译器会自动做这项工作的。

7.6.2 位逻辑操作符

C语言程序支持1个一元操作符和3个二进制按位逻辑操作符。每个这样的操作符都对存储在char, short int, int和long int数据类型中的值进行操作。

注意 二进制逻辑操作符执行操作数的数据升位以确保相同的精度。如果指定了一个short类型操作数和一个long类型操作数，编译器会把short类型操作数扩展为一个16位的long类型操作数，并且该表达式将会返回一个16位的整数值。

按位操作符AND（符号为&）对位于操作数中的每一对位都执行一个比特级逻辑AND（与）操作。例如，如果两个操作数中对应位都有值为0位，那么按位与的结果中该位的值将为0。

清单7-6 使用&的按位AND操作

```

int x=5,y=7,z; /*5 的二进制数是101, 7的二进制数是111*/
z=x&y; /*z 的值是5 (即二进制数101)*/

```

只要编译器具有允许使用二进制数据的扩展，那么AND操作是很容易做到的。

清单7-7 二进制值的AND按位操作

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x & y; /* z 得到的值是00000101, 或5 */
```

按位操作符OR（符号为|）对于操作数的每一对位都执行一个位级逻辑OR（或）操作。如果两个操作数中对应位任一方有值为1的位，那么按位或的结果中该位的值将为1。

清单7-8 使用按位OR操作符|

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x | y; /* z 得到的值是 00000111, 或7 */
```

按位操作符XOR（符号为^）对于操作数中的每一对位都执行一个位级逻辑XOR（异或）操作。如果两个操作数中对应位有一个为1，但不是两个均为1，那么按位异或的结果中该位为1。就是对操作数中没有共同值的位产生结果为1。

清单7-9 按位XOR操作

```
int x=0b00000101,
    y=0b00000111,
    z;
z = x ^ y; /* z 得到的值是00000010, 或2 */
```

按位操作符NOT（符号为~）产生二进制值的补码。操作数中每一个值为1的位的值都转为0，而每一个值为0的位的值都转为1。

清单7-10 按位NOT操作

```
int x=0b00000101,
    z;
z = ~x; /* z 得到的值是 11111010, 或250 */
```

如果将按位操作符应用到单个的位中，编译器使用位操作指令（如果可以用的话）。它将避免读或写到其他位时带来的没有预料到的副作用。

清单7-11 在单个位中的按位操作

```
void alternate( void )
{
    0300 0D 00 03  BRCLR  6,$00,$0306  PORTB.2 = ~PORTA.6;
    0303 15 01      BCLR   2,$01
```

(续)

```

0305 81      RTS
0306 14 01    BSET    2,$01
0308 81      RTS

```

7.6.3 移位操作符

位移位操作符的两个操作数都必需是整数值。

右移位操作符将数据向右移动指定位数。移动超过右边界的数据位将消失。对于unsigned型的整数值,如果必要的话,高端的那些0会被移动。对于signed型的整数值,移动的数值是与具体实现相关的。下面二进制数字右移number指定的位数。

```
X >> number;
```

一个二进制的数值右移 n 位相当于一个整数除以 2^n 。

左移位操作符将数据向左移动指定位数。移动超过左边界的数据位将消失,新移入的位为0。下面二进制数字左移number指定的位数。

```
X << number;
```

一个二进制的数值左移 n 位相当于一个整数乘以 2^n 。

清单7-12 向左和向右的移位

```

porta = 0b10000000;
while (porta.7 != 1) {
    porta >> 1;
}
while (porta.0 != 1) {
    porta << 1;
}

```

要移动可变的位数,可以通过在代码中建立一个循环结构来实现。但必须记住的是,这样会额外地占用一部分ROM空间。

清单7-13 移动可变的位数

```

00EB                                     int setting;

/* 根据从键盘输入的整数级设置LED位 */

0303 AD FB    BSR    $0300    setting = getch() - '0';
0305 A0 30    SUB    #$30
0307 B7 EB    STA    $EB

0309 A6 01    LDA    #$01    PORTB = 1 << setting;

```

(续)

030B BE EB	LDX	\$EB
030D 27 04	BEQ	\$0313
030F 48	LSLA	
0310 5A	DECX	
0311 26 FC	BNE	\$030F
0313 B7 01	STA	\$01

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

第8章 函 数 库

函数库包含用于一般用途和大范围开发工程所用的函数。嵌入式和桌面系统共用一些函数库（例如，增强型的数学运算功能或数据类型转换）。函数库是用于分类和传送这些特定知识的典型通用结构。

嵌入式系统更多地依靠函数库，一个函数库可以为一个普通的LCD控制器或外围定时器提供一个设备驱动程序。在一个嵌入式系统中，如果要对所有的事务负起全部的责任，程序员可能会被完全累垮。而如果有函数库协助直接操作外围硬件设备，程序员则可以轻松得多并专注在一个工程上。

由于C语言程序正向着一个高度可移植的方向发展，函数库是一个组织平台从属关系的途径。为一个特殊的8位微型控制器而专门编写的主程序C语言代码，只需要在代码上做一些小的修改就可以被编译，并且在不同的微控制器上运行。没有了函数库的可移植性，在一个特殊体系结构上的投资会增长，并且也降低程序员搜寻一种低价位处理器的兴趣。

Byte Craft代码开发系统公司研制了一定适用范围的实用可移植函数库（和传统的API形式的文档）。他们提供了8位嵌入式系统的一些最常用特性的例程。

- 标准 I/O

经过适当的配置，可以用一个键盘和一个LCD显示器作为标准的输入输出设备。

- SPI（串行外围设备接口）

- MICROWIRE 总线

- SCI（串行通信接口）

- UART（通用异步收发器）

UART是代替“bit banging”软件的主要候选者，该软件可被封装在一个函数库中。

- 模数转换和数模转换

- I/O端口

当操作I/O端口时，常常只是几个赋值指令的问题，但从特定实现中抽象出端口是很有好处的。

- LCD显示器

这些例程可以支持标准的I/O模式，并提供方便的例程用来清除显示器和移动光标。

- PWM（脉冲宽度调制）

- 定时器

8.1 创建函数库

本章将讨论如何从零开始创建函数库。

就拿自动调温器来说，我们需要显示当前时间和预先设定循环开始的时间，以一个字符串

的形式显示。时间串的长度为7个字节。

```
"12:00a" /* 带有一个尾符号a */
"06:35p" /* 以0作为前导符 */
"23:00h" /* 用于24小时制的时间表示方式 */
```

在以上的自动调温器中，我们实际上跟踪了4个时间：当前时间和3个循环开始时间。有多个二选其一的方式来存储这些数值，每一组都需要权衡。直接操作字符串是很难的：它会占用整整1/4的运行中的RAM空间，并且将会有许多代码用于临时进位和比较。

像分钟计数器（0~1439）那样的unsigned long型变量在ROM中证明是开销大的，但只占用8个字节的RAM（和中间结果暂存器）。用结构来实现时间计数器的组成部分（例如小时、分和上午/下午）很好，但用数组实现它们是不可能的。

两个整数构成的数组，一个是小时而另一个是分钟，看起来很不错。数组元素0是“当前时间”的一个很好的选择，元素1~3是为每日的“循环开始时间”准备的。

对于时间的文字表述，我们需要把它从一个时间计数器值（2个整数）翻译为一个时间戳串。不同的工程都会使用到这一类功能，所以我们要把它包装成一个函数库。经分析推断，24小时制和12小时制的系统都要能够支持，并且12小时制和24小时制之间的转换工作要能在运行的时候进行。

函数库应该具备两个函数：

```
void MinutesToTime( int hours int minutes );
void TimeToMinutes( int near *hours,int near *minutes );
```

以及两个变量：

```
bit use_metric; /* 定义转换格式 */
char buffer[7]; /* 执行转换的缓冲器 */
```

创建这样的函数库文件，应履行以下步骤：

- 1) 创建一个C语言源文件，命名为timestmp.c。
- 2) 写入下面的命令行。

清单8-1 源文件框架

```
#ifndef __TIMESTMP_C
#define __TIMESTMP_C

#pragma library;

#include <timestmp.h>

/* 上面已声明的：
    bit use_metric = 0;
    char buffer[7];
*/
void MinutesToTime( int hours, int minutes )
```

(续)

```
{
    ;
}
void TimeToMinutes( int near *hours, int near *minutes )
{
    ;
}
```

```
#pragma endlibrary;
```

```
#endif /* __TIMESTAMP_C */
```

3) 创建一个C语言头文件，命名为timestamp.h。

4) 写入必要的声明和原型。

清单8-2 头文件框架

```
#ifndef __TIMESTAMP_H
#define __TIMESTAMP_H

bit use_metric;
char buffer[7];

void MinutesToTime( int hours, int minutes );
void TimeToMinutes( int near *hours, int near *minutes );

#endif /* __TIMESTAMP_H */
```

5) 编辑C语言文件

```
c6805.exe timestamp.c +0 0=timestamp.lib
```

这是一个函数库的框架。当这个函数库建成时，将这个.lib文件与其他函数库一起存放，将这个.h文件与其他包含文件一起存放。

8.2 编写函数库

函数库软件和其他嵌入式程序非常相似。前面几章已经展示了哪些技术是安全的，哪些技术是昂贵，以及哪些技术在嵌入式系统的环境下是不可用的。

MinutesToTime() 接受一个表示小时的整数和一个表示分钟的整数。它检测use_metric标志，并把时间送到buffer[]中。

清单8-3 将小时和分钟转化为时间戳

```

void MinutesToTime( int hours, int minutes )
{
    char i;

    /* 设置字符串 */
    buffer[5] = 'h'; buffer[6] = 0; buffer[2] = ':';

    /* 处理12小时制时间 */
    if(!use_metric) {
        buffer[5] = 'a';
        if(hours > 11)
        {
            hours = hours - 12;
            buffer[5] = 'p';
        }
        if(hours == 0)
        {
            hours = 12;
        }
    }

    /* 填入小时 */
    buffer[0] = '0';
    for(i = '2'; hours >= 10; hours -= 10, i--);
    buffer[0] = i;
    buffer[1] = hours + '0';
    /* 填入分钟 */
    buffer[3] = '0';
    for(i = '5'; minutes >= 10; minutes -= 10, i--);
    buffer[3] = i;
    buffer[4] = minutes + '0';
}

```



另外，您可以展开for循环的底部来避开循环管理代码的作用。

TimeToMinutes()是一个逆向函数，它不能在自动调温器工程中使用。我们提到它是为它的简易和实用性。在自动调温器工程中，时间调节是由小时和分钟增加按钮完成的，与报警时钟非常相似。如果内存空间允许的话，相关配置将会被重写，以便用户能够用数字输入时间：用额外代码来检测那些输入的相对于有效时间的数字是十分重要的。

本应接收转换过的数值的TimeToMinutes()也接受指向小时和分钟整数值的指针。别忘

了它们是一种8个位数值的near指针。

清单8-4 将时间戳缓冲器的内容转换为小时和分钟

```
void TimeToMinutes( int near *hours, int near *minutes)
{
    if(buffer[0] <= '0') buffer[0] = '0';
    if(buffer[0] >= '2') buffer[0] = '2';

    *hours = (buffer[0] - '0') * 10;
    *hours += (buffer[1] - '0');

    if(buffer[3] <= '0') buffer[3] = '0';
    if(buffer[3] >= '5') buffer[3] = '5';

    *minutes = ((buffer[3] - '0') * 10);
    *minutes += ((buffer[4] - '0'));

    if(buffer[5] == 'p') *hours += 12;
}
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

8.3 函数库与链接

对于Byte Craft编译器，有两种函数库使用情况：带BClink的传统连接和绝对代码模式（Absolute Code Mode）。

在以前的说明中，timestamp函数库源文件是为绝对代码模式而编写的。如果要使用它们，就要按照下列所示编写主模块。

清单8-5 使用绝对代码模式的简单源文件

```
#include <705jla.h> /* 这里插入设备 */
#include <timestamp.h>

void main(void) {
    /* ... */
}


#include <timestamp.c>
```

要使timestamp适合于链接，需要在函数库的头部加入一些条件限定。理想上，头文件应同时既允许绝对代码模式，也允许传统链接。如清单8-6中的那样，使用MAKEOBJECT符号以便在

二者中作出选择。

将timestamp.h改变如下。

清单8-6 同时允许传统链接和绝对代码模式的头文件



```
#ifndef __TIMESTAMP_H
#define __TIMESTAMP_H

#ifdef MAKEOBJECT

#include <dev_def.h> /* 用CDS名字代替dev */
extern bit use_metric;
extern char buffer[7];

extern void MinutesToTime( int hours, int minutes );
extern void TimeToMinutes( int near *hours, int near *minutes );

#else /* MAKEOBJECT */

bit use_metric;
char buffer[7];

void MinutesToTime( int hours, int minutes );
void TimeToMinutes( int near *hours, int near *minutes );

#endif /* MAKEOBJECT */

#endif /* __TIMESTAMP_H */
```

如果timestamp.c本身就包含有头文件，就不需要再改变它了。

当创建一个函数库目标文件时，可以在命令行定义MAKEOBJECT。

```
Cds.exe -dMAKEOBJECT timestamp.c +0 0=timestamp.lib
```

这里，cds是编译器的可执行文件名。拷贝.lib文件到函数库目录中，并将.h文件拷贝到头文件目录中。

定义MAKEOBJECT符号将会导致函数和变量成为extern（外部的），并且将包含一个定义文件。所谓定义文件是指一个设备头文件，它带有所有重要的设备符号的定义（例如，端口、时间寄存器，等等）。最常用的值都已囊括在内，但这些并不重要：编译器使用定义文件编译函数库为目标程序，而不是依靠一个特定的设备头文件来实现。在链接期间，实际的设备值将会与目标文件中的引用相匹配。

当编译形成一个目标文件时(+0出现在命令行中),一些Byte Craft 编译器将会自动定义符号MAKEOBJECT。

另一个定制是很有用的:在RAM中,您也许想要用其他的方式来声明buffer[]是一个7字节的串(例如,像SPECIAL一样的内存)。如果使用绝对代码模式,可以用一个#ifdef来限定其声明。



第9章 优化和测试嵌入式C语言程序

正如在其他程序中所做的努力一样，把编好的代码进行编译仅仅是为了确保语句的正确性。在不理解编译器性能的情况下，我们无法阅读编译后生成的代码。

在不理解编译器限制的情况下，我们无法加入人的直觉知识。编译器无疑是减轻工作乏味性的最佳选择，但它们无法与充满灵感的程序设计工作相媲美。

测试嵌入式C语言软件与测试桌面系统软件有着显著的区别。一个新的中心焦点正在不断形成：嵌入式软件常常扮演着一个不容忽视的角色。对桌面计算机保护的失误可能花去用户几个小时的时间来修复，但嵌入式系统上一个软件上的失误则是危险的，包括：

- 对用户安全或身体舒适的影响。
- 对重要通信线的影响。
- 对主机设备的物理完整性的影响。

生命支持设备的问题不在本书涉及的范围之内。用于人体移植、监视或控制与健康相关因素的设备就是生命支持设备。编译代码是否应在这类设备上使用，这一问题仍是争论的焦点之一。编译代码的动机在于减轻不得不从头编写大量汇编代码的困难。生命支持系统不能随便冒这样的风险。

当选择一种处理器时，首先就要作出决定开发测试软件。有关开发用于微控制器的工具的更多信息请参见第3.7节。

9.1 优化

任何一个对编译器技术和科学感兴趣的人，很快就会明白优化是编译器编写者的永恒目标。关于编译器所能识别代码的任何引人关注的事实都会成为优化的候选对象。

尽管一些人认为艰苦的手工汇编代码是可以真正处理代码的惟一方式，但一个独立和客观的编译器可以找到另外一些适用于简约的隐蔽模式。

在嵌入式环境中，对优化的要求甚过其他环境。拿8位微控制器来说，成功的优化从根本上减少了使用ROM和RAM的数量。这是对生成代码的苛刻测试。不断增长的运行速度已进入了远远小于秒的量级。

这里有许多为优化生成代码的传统战略。编译器一直在观注着这些因素。

代数等值 如果对一个变量的引用导致它被装载到寄存器中，且紧近其后的是已知具有相同值的另一个变量的一个引用，则编译器会省略额外的装载操作。

寄存器数据流 如果一个变量将被装载到一个寄存器中两次，并删除冗余量，那么编译器会把它辨别出来。

冗余的或死的代码 从未证明真实的表达式所支配的代码在编译时可以忽略。跟随在break或continue语句之后的永不被执行的代码由于控制结构的不变性，它们将被废弃。

邻近指令的简约 简单指令的某种模式可以被简约后放入一个更复杂的操作中，就像带有自动增加功能这种副作用的指令一样。

合并常量 估计源文件中的常量的取值，合并取值相同的常量。

放样 如果一个循环结构中的指令不能直接适应此结构的话，可以把它放样到一个封装的语法层次中。

涉及低值常量的数学运算 0, 1, 2这样的操作数可以转化成类似增量或减量这样的结构，从而减少代码的大小和提高运行速度。没有一个代码是为加0、减0，或乘1、除1而产生的。

边缘效果 引起数值在它们的变量之间来回滚动的代码可以被视为一个需特殊处理的候选对象。

Long操作 在仅有一个8位寄存器的控制器中，long操作要花去多于指令两倍多的空间（一些控制器可以把一对寄存器转换为16位的变量，并用于long操作）。只要有一点儿关于可能数值范围的知识，就可以决定是否忽略16位变量的顶部字节或底部字节。

数组计算 针对一个数组元素的固定引用在编译的时候是通过间接访问的。这样就避免了重写一个变址寄存器。

与指令集相关的优化

由于指令集的一些特点，一些优化是可能的。

- 将加1变成递增，将减1变成递减。
- ++递增一个内存单元，--递减一个内存单元。如果变量是long类型的，则进位必须与后续指令一起被保留起来。
- 位操作通过使用位设置和位清除指令进行，而不是使用执行装载、按位AND或OR、存储的多字节序列。

9.2 手工优化

如果一个编译器负担着接受高级程序和产生优化机器语言的任务，为什么不考虑手工优化呢？考虑编译器的所有能力，编译器做不到观看“大型图像”。有时，它可以很好的跟随您的高级指示。

这里有一些节省ROM和RAM的策略。

检查寄存器的使用 在小的例程中，启动的包含有函数参数的寄存器也许并不使用，尤其是当这个例程直接操作内存时（例如，带有专门指令的二进制数字处理）。通常我们的回应是声明像int一样的函数参数，在大多数情况下，这会导致局部RAM来存储这一数值。把一个函数参数声明为寄存器类型（在Byte Craft编译器上是registera或其他等价类型）可以节省字节。

循环结构的滚动或展开 展开一个易懂的短循环结构看起来似乎是不直观的，但节省ROM空间会使它显得有利可图。寻找机会的过程也就是针对循环结构状态和动作部分生成昂贵代码的过程。

把端口当作变量来用 不要低估嵌入式语言程序员在使用RAM过程中为追求节省的执着精神。如果一个输出端口可以被安全地读取，以确定输出针脚的当前状态，并且这个端口需要一个循环结构操作，那么就没有任何理由不把端口本身当作一个索引变量来使用。考虑以下情况。

清单9-1 把端口当作变量来使用

```
#pragma portrw PORTA @ 0x00;

void walk_through_A(void)
{
    for(PORTA = 0x01; PORTA != 0; ASI(PORTA))
        delay_100us(10);
}
```

在这个例子中，如果一个独立的char变量用于给循环索引，并给端口赋值，那么就没有理由认为编译器可以忽略其他不用的变量。虽然编译器会考虑端口的易失性，但我们可以从这个设计中确定，在这一具体情况中端口是否会以易失的方式活动。

人工变量调整

在一个传统的C语言环境中，编译器可以在没有过多人工干预的情况下进行变量分配。例如，在一定范围内为每一个计数器变量名分配一个新位置是一件很普通的事。

清单9-2 局部计数器变量

```
void up_and_down(void)
{
    int up, down; /* 可能是分隔的两个位置 */

    for(up = 0; up < 128; up++)
        porta = up;
    /* ... */
    for(down = 127; down > 0; down--)
        porta = down;
}
```

为了最小化RAM的使用，嵌入式系统开发人员会经常创建全局循环计数器变量。当需要一个计数器或一个临时变量时，任何一个函数都可以使用这个已分配好的数据内存块。程序员会监视封装的循环结构之间的冲突。

一种解决办法是严格声明这些变量为局部的：一些C语言的编译器支持固定一个符号在内存中位置的功能扩展。您可以利用这一特性来管理变量如何被置于数据内存空间。这里有对于Byte Craft编译器的合适注释。

清单9-3 局部计数器变量覆盖另一个变量

```
void up_and_down(void)
{
```

(续)

```
int up;
int down @ up; /* 覆盖 */

for(up = 0; up < 128; up++)
    porta = up;
/* ... */
for(down = 127; down > 0; down--)
    porta = down;
```



由于声明非常特殊，编译器将按其要求执行。就已被分配变量空间的再利用，而不用求助于宏或其他技术这一点来说，这样做是很有用的。如果内存开放，则只有@location扩展文件需要被删除。

9.3 调试嵌入式C语言程序

在学习了如何解释编译器代码生成的结果后，可以开始对嵌入式C语言程序进行调试。

在嵌入式系统中，对C语言程序进行调试是存在一些弊端的。

9.3.1 寄存器类型的修饰符

那些执行register关键字的编译器也许不会真正授予访问存储器的独占性。8位的MCU（微程序控制器）不会有很多的寄存器来分享。相反，编译器也许会从可用的最快的内存中分配。

其他关键字，像Byte Craft 的registera和等价关键字，将把一个标识符同适当的寄存器关联在一起，但结果变量被认为是易失的。您可以立即访问在系统中使用的所有汇编代码。通过访问，您可以用检测的方式来决定编译代码是否干涉了寄存器的内容。

9.3.2 局部内存

如果编译器支持局部范围内的变量值，应确定编译器为函数调用中的变量分配内存的方式。

以下有三种局部内存分配的策略：

在栈帧中 这要求有一个很清楚的栈相关的寻址策略，这无疑是很奢侈的。它并不总是首选的代码选择，即使这样是可行的，编译器也未必会选择使用它。

从全局堆中 变量只是简单地从需要的RAM中分配出来。全局和局部混合在一起。

“专注于”局部内存 用于和重用于多个函数调用内部。

9.3.3 指针

因为哈佛体系结构的MCU有两个由上下文来选择的地址空间，指针必须指向程序（ROM）空间或数据（RAM）空间。这可能导致代码次序的混乱。

在一些结构中，far指针变量只能由自修改代码来完成。要了解更多的信息，请参见9.6节。

9.4 混合C语言和汇编语言

嵌入式系统代码所存在的环境远比传统的应用软件所需的环境严格得多。直接向汇编代码求助是不可取的，除非必须要固定记时观察，或在当前的工程中使用先前存在的汇编代码。

9.4.1 调用规范

嵌入式C语言的交叉编译器为调用函数生成了不是很标准的代码。当调试程序时，应知道以下一些问题的答案。

- 编译器是否设置了页位，或实行存储体切换，来优先调用子程序？
- 在中断期间，编译器或处理器是否保存和恢复状态？
- 函数自变量是怎样被传递的？结果是如何返回的？几乎可以肯定的是，一个8位结果将会被留在累加器中。

9.4.2 从汇编代码中访问C变量

汇编代码能够准确地寻址C语言标识符吗？虽然编译器允许把C语言标识符当作一个自变量在汇编存储器中使用，它也可能不会检测该数值的大小，虽然指令有预先规定的大小。结果导致程序载入多字节数值中的一个字节，而不去考虑它的实际意义。

9.5 试验硬件

如果您有权使用目标硬件的原型，一个用来测试硬件的小程序将会坚定您对它的配置和性能的信心。

如果主要工程不是像预计的那样运行在一个仿真或发展的系统中，同样的技术将会决定问题是发生在软件上，还是在硬件上。

9.6 通过检查调试

编译器可以帮助通过生成不同形式的报告来检查代码。Byte Craft编译器集中了在清单文件中的所有报告，清单文件集中了生成的代码和源代码。正如在9.2节中讲述的那样，这些报告会有助于人工优化的琐碎工作。

编译器应生成一种它所认识的所有符号的映射。由Byte Craft编译器生成的符号表在以下的清单9-4中有格式化的说明。

清单9-4 符号表摘录

符号表			
标 签	值	标 签	值
CC	0000	COPC	0000
COPR	07F0	DDRA	0004

(续)

DDRB	0005	IRQE	0007
IRQF	0003	IRQR	0001
ISCR	000A	LOCAL_START	00EB

以上列出的符号同时声明了变量、函数和预处理程序符号。由其他方式声明的标识符，如 #pragma 语句，也出现了。这是编译器能明白的所有标识符的详细目录。

桌面程序员们不经常处理指针的实际值。通常他们把一个目标的地址赋给一个指针变量，然后操作这个指针（加1或减1）。指针的实际数值最好是处于未知状态，因为它是将要改变的。

由于在一个8位嵌入式系统中代码和变量将不会被重新部署，并且由于RAM是宝贵的，因此在嵌入式环境下检测RAM的分配是很有用处的。

清单9-5 RAM使用映射的摘要

RAM使用映射

0050 use_metric	signed char		
0051 buffer	unsigned char[6]		
0000 CC	register cc		
0000 PORTA	portrw		
0001 PORTB	portrw		
0057 temp	unsigned long	0100	0114
0051 buffer	unsigned char[6]		
005D hours	unsigned char	011A	01DE
005E minutes	unsigned char	011A	01DE

这一报告显示出了全部符号，每个符号都有其值分配的内存和各自的位置。这是由 &（即取其地址）操作符返回的位置。局部变量与其所处的程序范围一起被列举出来。

编译器会给出全部的内存使用计数。这是对程序员和编译器的严峻考验：不同的代码是否可以传递，不同的理论或者不同的优化方法是否可以节省几个额外的ROM字节？

程序清单本身可以被定制。为了方便起见，编译器会把每个操作数的执行时间列出一张表来。您可以计算它们来完成一些工作，例如，估量出一个中断服务例程所需的运行时间。这一信息接下来可有助于校准依赖于定时的函数。

在Byte Craft编译器中，一个有用的清单文件选项描述出每一个C语句块的嵌套级别，正如编译器所了解的那样。一个相似的选项在单独的报告中揭示出函数调用的层次。

```
#pragma option NESTINGLEVEL;
#pragma option CALLMAP;
```

CALLMAP最有用的一个方面就是决定使用了多少栈。编译器为栈设置了一个静态深度。运用CALLMAP程序和您对系统的知识来定制栈的大小以节省未使用的空间。

编译器也可以显示出它所知道的保存在处理器寄存器中的值。如果是在没有仿真器程序的

情况下工作，这将提供给您一些仿真器程序将追踪的信息。

9.7 假载荷

检测微型控制器软件的方式之一就是引导控制器在一个假载荷的环境下进行操作。这是一个比软件工作更繁琐的硬件技术，它的要点是用简单的按键、继电器、指示灯来复制目标系统的每一个外部元件。利用对目标系统运行方式的知识可以重建控制器所期望的信号，并随时观察控制器的反应情况。

9.8 仿真器和模拟器的运用

程序编译完之后，必须用仿真器和模拟器来检测。

9.8.1 模拟器

模拟器是一个基于主机或桌面的软件应用程序，用于评估为嵌入式目标机器而设计的程序。模拟器重建目标机器的运行条件并解释执行情况。

使用模拟器时，可以在程序运行时一步一步地执行代码。模拟器会报告寄存器和状态值、外围设备寄存器的内容以及RAM的使用情况。

由于模拟器不是基于硬件的，所以缺少物理电气设备的特殊特性。模拟器可以根据微处理器文档来编写，因此会忽略任何硬件内部制造细节。

9.8.2 仿真器

仿真器是一个硬件设备，它的电气和逻辑行为与目标处理器十分相似。也许它会包含一个相似的处理器，但却是由额外编程来支持研制中的主机的控制和通信的。仿真器与实际研制系统之间有一个链接，它在检测状态下提供一个进入设备的窗口。由于微控制器通常包含系统所需的ROM和RAM，因而它们一样处于外部控制之下的。

当来自预定产品版本接受检测的程序未改变时，仿真器工作状态是最好的。尽管对以下将解释的理由来说，这并不总是可行的。

普通仿真器的特性概括如下：

- 设置断点的性能

好的仿真器是在一个“额外”地址表的基础上设置断点的。当仿真程序运行到这一位置时，断点中止程序执行，并等待用户的干预。

另一个做法是重写程序：一个仿真器可能会存储断点位置的值，并把它写入一个软件中断指令中。软件中断会调用那些将控制返还到仿真器主机中的管理代码。

- 支持、改变寄存器和内存单元

一旦处于一个断点中，仿真器会无损地报告目标处理器的内部状态。

- 跟踪缓冲器以分析总线的通信量

当并不是直接与软件相关的时候，一个昂贵的仿真器会给出关于目标处理器电气和定时信号的详细信息。

借助仿真器来进行调试和检测，一个特殊的挑战就是频繁的中断调用。中断如果发生得过于频繁或总是存在时间很短的话，会很容易造成仿真器搭接。只有带有扩展跟踪缓冲器的高端仿真器才可能准确地记录这些事件的执行过程。

其他挑战来自于半导体封装技术的进步。在电路中仿真器需要连接一个目标系统以代替微控制器。MCU封装技术已经从DIP型缩小到微小的表面组件部分。所要求的稳定的物理连接对于工程师来说难度在不断增加。

关于外部仿真器问题的讨论是值得的。专用的硬件是低容量、高复杂性的，因此是昂贵的。仿真器处理MCU的外部信号：它们也许靠牺牲速度来换取一个简单的操作技巧，或通过成本和复杂性的极大增长来提供一个实时的信号仿真和监视。

有以下两种方式来解决成本问题。

- (1) 用ROM仿真器来取代在目标系统中的外部程序存储设备，与代替微控制器的仿真器相比，这种做法的复杂度较小。通过返回程序的操作数对指令捕捉作出回应，并且可以在任一点上插入软件中断。另外，它也能提供监控器代码，该代码是目标微处理器操作断点所需要的。
- (2) 许多新的MCU设计正在将芯片级仿真功能一体化集成到每一个生产设备中。这里的主要目的是建立一个完全的原型，该原型有永久现场的正规样本或生产处理器。比使用一个特定的仿真设备更合理的做法是，开发者可以使用内置的仿真功能来考验处理器。到控制主机的链接是由一个2~4针的串行接口提供的。在此原型中，仿真信号被发送到一个头信息条中，并且一个小的电缆和插孔也可以通过一个串行接口提供到主机的链接。最终的设计也许不会实现此头信息条的特征，除非需要使用此方法提供对现场工程师的访问。这样的跟踪不会有任何担心。

9.9 嵌入式软件的封装

一个嵌入式程序通常被编译为一种专用的十六进制或二进制的表示形式。这样的输出适用于下列情况。

- 下载到编程设备上

对于检测 and short 类型运行来说，可编程ROM的个别芯片也许有集成在其中的、由编译器创建的二进制映像。

- 向屏蔽的芯片产品提交信息

对于long类型运行来说，一个制造设备可以编写二进制的信息到用于硅制品的封包中。每一芯片的构成都带有ROM单元集，此单元集符合上述二进制映像。

第10章 样例工程

本章内容是关于自动调温器工程技术主题的，不是先前已讨论过的内容。

自动调温器的源代码在本书所附带光盘上有。如果您想构建自动调温器，在光盘上可以找到详细的信息资料。本章详细说明了几项技术主题，这样的讨论对其他工程上也同样有益。

更新和修正的信息可以通过以下网址获得：

http://www.bytecraft.com/embedded_C/

10.1 硬件的练习程序

这些程序过去曾用于检测自动调温器的硬件。我们编写它是为了认识印刷电路板可能带来的挑战。要使用C和JICS仿真器进行试验，它们是有关输入和修改的很好例子。

10.1.1 显示“Hello World!”

由于我们没有自动调温器电路板上的LED（发光二极管）指示器，所以要锁定制热/制冷单元继电器中的一个进行观察。LCD函数库还没有设置好。

清单10-1 通过继电器的“Hello World!”

```
#pragma option s5; /* jics的映射文件 */
#pragma option f 0; /* 在清单文件中没有页间断 */          */

#include <705j1a.h>
#include <port.h>

unsigned long counter;

void pause(void)
{
    for(counter = 0; counter < 255; counter++)
    {
        NOP();
    }
}

void main(void)
{
    }
```

(续)

```
PORTB.0 = 0;
DDR_MASKED(PORTB, _____C, 00000000);
DDR_WAIT();

while(1)
{
    pause();
    PORTB.0 = 1;
    pause();
    PORTB.0 = 0;
}
```

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

10.1.2 键盘测试

下一步将做键盘设定。依据硬件设置，键盘函数库也许会要求用户定制。在以下例子中，它需要一些修正。

清单10-2 键盘测试程序

```
#pragma option s5; /* jics中的映射文件 */
#pragma option f 0; /* 在清单文件中没有页间断 */

#include <705j1a.h>
#include <delay.h>
#include <port.h>

#define KEYPAD_PORT PORTA
#define KEYPAD_DDR_REGISTER DDRA
#include <keypad.h>

void main(void)
{
    int8 store;

    /* must keep LCD_E low */
    PORTB = 0;
    DDR(PORTB, 00000000);
    DDR_WAIT();
```

(续)

```

keypad_init();

while(1)
{
    switch(keypad_getch()) {
        case '0'      : PORTB.0 = 1; break;
        case '6'      : PORTB.0 = 0; break;
        case '#'      : PORTB.0 = ~PORTB.0;
    }
}

#include <keypad.c>
#include <port.c>
#include <delay.c>

```



10.1.3 LCD测试

这是一个用来测试LCD（液晶显示器）的简单例子。

注意LCD函数库所需要的配置。其符号和可能的数值在函数库参考资料和lcd.h文件中有。

清单10-3 LCD测试程序

```

#pragma option s5; /* jics的映射文件 */
#pragma option f 0; /* 在清单文件中没有页间断 */

#include <705j1a.h>
#include <delay.h>
#include <port.h>

#define LCD_DL 0
#define LCD_UPPER4 1
#define LCD_DATA PORTA
#define LCD_RS PORTB.2
#define LCD_RW PORTB.3
#define LCD_E PORTB.4
#define LCD_CD DDRB
#define LCD_CDM __CCC__
#include <lcd.h>
void main(void)
{

```

(续)

```
lcd_init();

while(1)
{
    puts("Hello World");
    delay_100us(10);
    lcd_send_control(LCDCLR);
    delay_100us(10);
}

#include <lcd.c>
#include <delay.c>
```



10.2 与端口通信

与函数库一起工作的最大挑战就是，当共享端口时确保它们之间彼此配合。如果函数库不能完全控制它所需要的端口，更重要的是不能把它们保留在一个稳定的状态下，则您就会有错误驱动外围设备的危险。

典型的基本字符的LCD界面使用

- 8或4根数据传送导线。
- 一个用于指令选择或数据选择的导线。
- 一个用于读或写的导线。
- 一个用于启用的导线。

在自动调温器设计中，LCD显示器的数据导线是拥有键盘矩阵的四根导线的多路系统。

以下是我们设计的保持键盘和LCD组织路径的方针。

- 在读或写数据之后，确保LCD的启用操作线设定为禁止状态。这项工作由快速代码检测来完成。
- 确定端口方向设置要求的例程。lcd_read()和lcd_write()函数要求数据方向设置，正如他们实际驱动LCD接口一样。其他如lcd_set_address()这样的函数库例程使用这些函数，因此它们不需要自己特有的端口方向设置。

尽管keypad_getch()使用keypad_kbhit()，但它们都需要数据方向设置。keypad_kbhit()的意图在于方便用户自己设置的轮询循环结构，而keypad_getch()一直要等用户按一个键才返回。

10.3 A/D转换器原理

这个设计的特色是有简单的A/D(模/数)转换器电路，以代替在第3章中介绍的专用外部转换器。由于取消了对集成A/D外围设备的要求，因而增加了供选择部件的数目。

这种设备的主要特点之一就是它便宜。对于大量生产的设备来说，这是一个很重要的考虑因素。其折衷结果是做一个集约软件。

下图就是电路，注意 R_i 是一个热敏电阻。

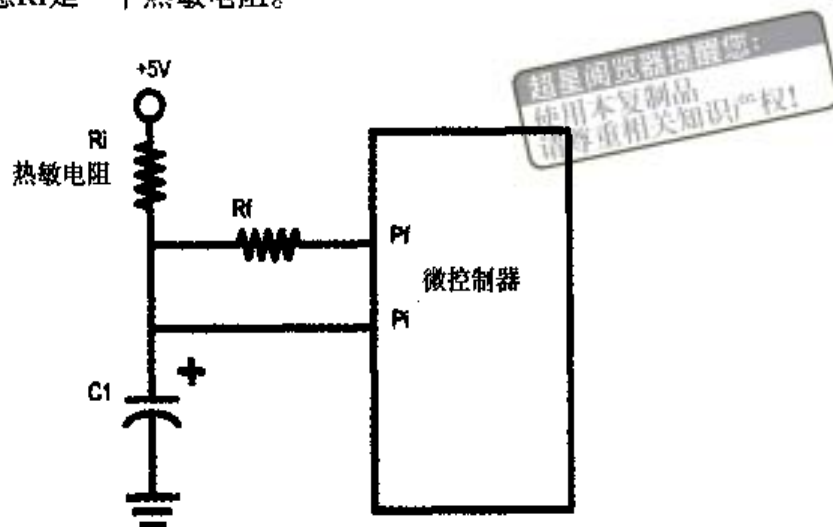


图10-1 A/D转换器电路

A/D转换器假设嵌入式微处理器端口的输入阻抗相对较高，且开关点始终保持较小的滞后。

同样假设 R_i 和 R_f 之间的接合点是一个电流汇集点，具有电容器 C_1 积分误差电流的功能。通过发送一组流出端口位 Pf 的脉冲，微处理器可以调制经过 R_f 的电流。某一次的全部数目和全部发出的位数的比率是在 Pf 上的平均电压的函数。可以把微处理器作为一个高增益运算放大器来看，它试图保持在汇集点的电压，该电压为 Pi 点由低到高感应电压的阈值。

从物理上说， Pf 是PORTB的位5， Pi 是IRQ（中断请求）的输入，作为一个中断源被屏蔽。当 Pi 加锁时它必须被复位，但在其他方面它像一个输入位。想要获得A/D转换器输入范围的概念，只需在自动调温器上运行下列代码。

清单10-4 简单的A/D驱动程序代码

```
#include <705j1a.h>
#pragma msc @0x7F1 = LEVEL;
#include <port.h>

#define Pf PORTB.5
#define Pi ISCR.IRQF

void main (void)
{
    DDRB = 00000000;
    Pf = 0;
    ISCR.IRQE = 0; /* 请不要中断 */
    ISCR.IRQR = 1; /* 复位IPQF/Pi准备启动 */
}
```

(续)

```

while(1)
{
    Pf = Pi; /* 如果使用普通位进行输入, 那么反转 */
    if(Pi) ISCR.IRQR = 1; /* 复位Pi锁 */
}
}

```



通过调节针脚Pf的范围来加热或制冷热敏电阻。

这一模式实际上把微型计算机当作一个高增益的运算放大器来使用。范围会显示一个脉冲流, 其工作周期将和来自Ri点的输入电压一致。在范围踪迹中的零点与全部时间的比率是输入电压的一个正函数。就是这个比率使我们最终想利用软件来测量。

可以测量出的输入电压的范围取决于下述因素: 输入端口的读出电压(Vs), Pf点在高和低状态的输出电压(Vh和Vl), 以及电阻器Ri与Rf的电阻值。下例方程式确定可以被A/D转换器读取的最低和最高输入电压。

$$V_{min} = (V_h - V_s) * (R_i / R_f)$$

$$V_{max} = (V_s - V_l) * (R_i / R_f)$$

当Pi始终恰好保持在读出阈值上时, 并且处理器总是反馈一个1到Pf针脚时, 会出现Vmin值。当出现一个Vmax输入时, 从Pf中会始终反馈一个0。在最高与最低电压值之间, A/D数值是线性的和成比例的。这取决于一次测试实例中, 在Pi点获得1的次数(N1)与全部测试次数的比率。系统的准确性是测试样本总数(N)的线性函数。Vi可以由下列关系来计算。

$$V_i = (N1 / N) * (V_{max} - V_{min})$$

C1数值不是临界值, 它用来控制系统的转换速率和抗噪声度。对于一个典型的系统来说, 测量从0~5伏特的一个输入, 是从一个47K的电阻器和一个0.01~0.1微法拉的电容器来开始的。

最后, 类似这样的比率度量测试系统提供转换精度——精度是转换时间的函数。而且, 其结果可以容易地适应该应用。这样消除了由改变样本大小而建立的转换乘法。

附录A 嵌入式C语言函数库

使用本复制品
请尊重相关知识产权!

A.1 引言

为进一步减少开发成本，自然要大力促进硬件和软件产品的标准化。标准化的计算机导致标准化的语言开发和标准化的操作系统。同时，开发人员建立了使用广泛的常用标准函数库。

与之形成鲜明对比的是，关于8位嵌入式系统的普遍观念是：每一个新的设计都是一次性的编程任务。大量的应用并没有使它成为标准硬件。只是最近几年，编译器相当或超过了手工生成汇编代码的效率。最后，程序设计的内部级别禁止对第三方软件作出种种假设。

我们的经验是：8位系统的编程可以借鉴促进主流计算机系统发展的研发实践工作。虽然体系结构是多种多样的，但从功能上说嵌入式系统硬件仍是标准化的。例如：I/O工具有端口针脚的特性，如可选的三态，但在排列的限制数目之内。并且，控制器常使用高度标准化的总线，如SPI或CAN；尽管接口不同，但期望的结果仍是相似的。

这些硬件上的相似之处导致了开发语言的标准化。我们还发现大量的嵌入式应用可以用C语言实现，并且可编译成适用于不只一种的当前市场上处于领导地位的微控制器体系结构。正如桌面计算的发展一样，选择标准的开发语言会减轻您对特定结构和提供商的依赖，从而能够减小成本方面的压力。

余下大量未经考察的问题是用于8位环境的C函数库是否可以标准化。对于嵌入式系统，它能否扮演与在桌面计算机软件环境开发中相同的角色？这一设想是十分吸引人的。

缩短了进入市场的时间 这是跟踪每个产品的一个简单积累。函数库代表已经走过的必需的步骤。

可重用的代码库 函数库代表预先消化的知识，代表一个非常了解、良性结构、很好档案化的代码体。这样做的回报是用更短的时间和更少的精力就能满足定制和配置这些代码的要求。在C语言中，所谓配置就是用#define指令来回答一些问题。

产品的可靠性 每一个重新使用函数库的开发工程可以再次考察函数库以求质量保证。由于每一个函数库的用户都可以访问源代码，因而局部化的定制和修改可以集成到函数库中为以后的人使用。

当然，再往下的挑战是将大量没有预见的应用也调和到一个权威标准中。

利用函数库工作是没有问题的。执行乘、除或取模运算的软件最好作为函数库的外部集合提供，这个函数库由编译器在需要的时候读入。但是，关于预定功能的设计有一个小小的争论，即是否存在一些操作符，在任何情况下它们都有共同的调用接口。

接口的表述是一个最大的障碍。扩展的数学运算和外围设备功能对标准化的功能接口和函数库实现的要求最为迫切。浮点运算、8比特实现权衡、逻辑除运算是最可能的争论焦点。

具有挑战性的工作是找到鲁棒性强的、适应嵌入式特殊要求的通用接口。

有效的函数调用 带有很少栈空间的8位体系结构并非函数调用的候选方案。函数库的形参对一些用户而言，总是包括一个多余的值。

现在假设仅有一个编译器在工程上工作，那么函数实施的物理部分就是确定的。编译器可以做任何事情来克服目标设备的资源限制。

逻辑上相似函数的物理差别 输入和输出位可能都表示在I/O针脚上的实际电压，但对于数据方向设置并没有取得一致意见。

C语言可以很容易地适应符号的改变，参考一个理想抽象的端口函数库。

外部设计选择 这个问题实际上并不容易解决。如果在一个端口上有两个外围设备是多路复用的关系，如在自动调温器那种情况下，那么它们之间会相互影响，而这一点函数库可能预先没有估计到。C语言可以很容易地适应多级符号的改变，但对设计的挑战却由设计的复杂性变为对设计的不可捉摸。

为什么说把函数库源代码和编译器放在一起是十分重要的，这最后一点是一个原因。前面已讨论过的产品可靠性是另一个原因。幸运的是，当代软件的工业实践，从经济的角度来看，允许甚至鼓励源代码的这种分布式状态。Byte Craft很早就认识到了把函数库源代码和编译器放在一起的重要性。

以下小节概要讲述一个强健的标准函数库接口。这里，函数库是有用和可移植的。在需要的地方，我们已经尽可能地服从于桌面C语言函数库的接口。

A.2 使用函数库

按照下述步骤可以很容易地使用函数库。

- 添加include子目录到您的环境的INCLUDE环境变量中（完整的路径名取决于您的配置）。或者，在带有n=命令行选项的命令中指定include子目录。
- 添加lib子目录到您的环境的LIBRARY环境变量中（完整的路径名取决于您的配置）。或者，在带有t=命令行选项的命令中指定lib子目录。
- 使用#include <>语句在源代码的起始部分加上带有#include <>语句的头文件。例如：

```
#include <stdio.h>
/* 您的主函数和其他代码 */
```

这涉及到绝对代码模式的编译器手册。编译器为源代码起始部分的每一个头文件搜索相匹配的函数库文件。

设备头文件和定义文件

代码开发系统依赖于含有有关定义和常量的头文件。这些头文件依功能部件数目的不同而不同，它们通常以其提供的功能部件来命名，并带有.h扩展名。

更多的信息参见A.4节。

A.3 数学函数库

代码开发系统的数学函数库包含在一个文件中，该文件的文件名与产品的名称相匹配。该文件常以源代码的形式提供，但带有.lib的扩展名。这样，编译器就能读它，并在需要的时候编译。

数学函数库提供实现8位或16位数值进行*、/和%等操作的功能。相关的函数名称如下。

操作符	函 数
*	__MUL8x8(void) __MUL16x16(void)
/	__DIV8BY8(void) __LDIV(void)
%	__RMOD(void)

超星数字图书馆
使用本复制品
请尊重相关知识产权!

如果想调整其中的数学程序为您喜欢的样子,那么备份函数库文件,然后直接对其进行改变。例如,对于名为ABC的代码开发系统(Code Development System, CDS)产品,它的数学函数库的名称也将是ABC.LIB。

并不一定非要在主程序中#include此函数库,因为需要时编译器将自行包含它。编译器将在下述目录查找函数库:

- 在当前目录下。
- 在LIBRARY路径指明的目录中。

因此,在程序的LIBRARY路径中要有Byte Craft函数库子目录,这一点是十分重要的。

A.4 函数库定义

DEF.H

注意

存在多种CDS产品,定义头文件的名称也将随之改变。在名为abc_def.h的文件名中,abc是一种CDS产品的名称。

描述

定义头文件对于编译函数库是十分有用的。

在编写公共代码的函数库时,可能并不知道哪些目标芯片要编译。如果不包含设备头文件,就不能使用标准的标识符来编写代码,这些标识符会使程序更易于阅读和维护。

解决这一困境的办法就是包括函数库定义头文件代替任何一种特殊设备头文件。函数库定义文件定义了出现在每一个设备头文件中的所有的标准标识符。

当将函数库编译成目标文件的时候,Byte Craft编译器将忽略在定义文件中的定义值,而仅保留标识符。在进行链接的过程中,编译器将把这些标识符链接到在特殊设备头文件中指定的实际标识符。

例子

这个例子假设使用绝对代码模式(也就是说,不使用BCLink)。如果将函数库同BCLink链接,要正确地声明函数库功能为extern。MAKEOBJECT的定义可以帮助决定有条件地去做。

当编写名为my_library.lib的函数库时,包括def.h头文件。

```
#pragma library
```

```
#pragma option +l /* 在清单中保留函数库代码 */
#include <abc_def.h>

void my_func1(void)
{
    PORT0.1 = 0; /* 在abc_def.h中使用一般定义 */
}

#pragma endlibrary
```



将上述文件编译成目标文件，将此目标文件重新命名为带有.lib扩展名，然后将它存在LIBRARY路径指定的目录中。

建立一个函数库头文件。

```
void my_func1(void);
```

将这个文件以my_library.h保存在您的INCLUDE路径指定的目录中。

建立程序源文件，并在其中包括设备头文件和函数库头文件。

```
#include <specific_device.h>
#include <my_library.h>

void main(void)
{
    /* . . . */
    my_func1();
    /* . . . */
}
```

像平常一样，编译这一程序源文件。

STDIO

1. STDIO.H和STDIO.C

名称

stdio是一个标准的输入和输出函数库。

描述

stdio能使C语言更容易地编写嵌入式系统的程序。虽然带有流的操作系统在8位微处理器上通常不适用，但程序员仍可以调用一些熟悉的函数来实现对预知设备的输入和输出操作。

stdio还能提供更复杂功能的嵌入式系统表述。由简单的分析可知，scan()函数可从用户提供的getch()读取字符，并且评估对应于缓冲器中模板字符的键码（对于数字是“0”，对于字母是“a”，等等）。一个实验性的实现大约消耗200字节的ROM空间。

2. gets和puts

名称

gets()和puts()是一个输入和输出字符串。

语法

```
#define BACKSPACE...  
#include <stdio.h>  
void puts(char far * str);  
void gets(char near * str, int8 size);
```

描述

puts()输出以“空”字符结尾的字符串给一个设备，该设备是这一标准输出语句所知道的。

gets()从一个设备接收一行字符，并且把它放在缓冲器中长度为size的str中，该设备是这一标准输入语句所知道的。其接收字符停止的条件是：出现新的一行，打字设备的托架换行，或size的值为size-1。缓冲器的最后一位赋值为零。

定义字符BACKSPACE，当getch()接收到BACKSPACE时，允许gets()回退。gets()实际上使用BACKSPACE完成回退操作，因而getch()设备必须提供BACKSPACE，并且putch()设备必须明白BACKSPACE是将输入点或光标回退一个空格的字符。

这些例程依赖于函数库中的函数getch()和putch()，它们必须在另一位置被声明。getch()和putch()可能的定义有：

- 在keypad函数库中的keypad_getch()。
- 在lcd函数库中的lcd_getch()和lcd_putch()。
- 在uart函数库中的uart_getch()和uart_putch()。

STDLIB

1. STDLIB

名称

stdlib是标准函数的函数库。

描述

stdlib包含有大量有用的函数。

2. rand和randmize

名称

rand()和randmize()生成伪随机数。

语法

```
#include <stdlib.h>  
  
#define SEED 0x3045 /* 种子不必是0 */  
#define srand(SEED) Rand_16=SEED  
#define randmize() Rand_16=RTCC
```




```
int16 rand(void);
```

描述

rand()提供和管理伪随机数序列。

randmize ()初始化伪随机数序列。

为了初始化伪随机数序列,在初始化子程序中调用randmize ()。然后,为每一个新的随机器调用rand()。

当前随机数存储在一个静态持续的数据对象中,并且在每一次调用rand()时都进行修改。

要求

部分头文件或定义文件,以及字符串函数库。

3. abs和labs

名称

abs()和labs()求出绝对值

语法

```
#include <stdlib.h>
int8 abs(int8 i)
int16 labs(int16 l)
```

描述

abs()接受带符号字值,返回其绝对值作为正的带符号字值。

labs()接受带符号的int16值,返回其绝对值作为正的带符号的int16值。

4. ui16toa, ui8toa, i16toa和i8toa

名称

ui16toa, ui8toa, i16toa和i8toa将无符号和带符号的整数值转换为ASCII的表述形式。

语法

```
#include <stdlib.h>
void ui16toa(unsigned int16 value,char near * str,
             unsigned int8 radix);
void ui8toa(unsigned int8 value,char near * str,unsigned int8 radix);
void i16toa(int16 value,char near * str,unsigned int8 radix);
void i8toa(int8 value,char near * str,unsigned int8 radix);
```

描述

ui16toa()将无符号的int16整数值转换为以“空”字符结尾的ASCII串。它接受指向字符串缓冲器的指针,数值被转换为字符串表示形式,并且其中的基数表示这个数值。

基数可以是下表中的数值之一。字符串缓冲器必须足够的长,以便包含转换所建立的所有字符。因此,缓冲器必须有对应的大小。



基数	表示形式	要求的缓冲器大小
2	二进制	16个字符
8	八进制	6个字符
10	十进制	5个字符
16	十六进制	4个字符

除了转换无符号的字值(8位), `ui8toa()` 与 `ui16toa()` 相同。因此, 输出缓冲器要求的空间如下。

表示形式	要求的缓冲器大小
二进制	8个字符
八进制	3个字符
十进制	3个字符
十六进制	2个字符

`i16toa()` 将带符号的 `int16` 整数值转换为以“空”字符结尾的ASCII串。它接受指向字符串缓冲器的指针, 数值被转换为字符串表示形式, 并且其中的基数表示这个数值。

基数可以是下表中的数值之一。字符串缓冲器必须足够的长, 以便包含转换所建立的所有字符。另外, 负值用减号(-)表示。因此, 缓冲器必须有对应的大小。

基数	表示形式	要求的缓冲器大小
2	二进制	16个字符
8	八进制	7个字符
10	十进制	6个字符
16	十六进制	5个字符

除了转换无符号的字值(8位), `i8toa()` 与 `i16toa()` 相同。因此, 输出缓冲器要求的空间如下。

表示形式	要求的缓冲器大小
二进制	8个字符
八进制	4个字符
十进制	4个字符
十六进制	3个字符

5. `ahtoi16`, `ahtoi8`, `atoi16`和`atoi8`

名称

`ahtoi16`, `ahtoi8`, `atoi16`和`atoi8`将一个ASCII串值转换为一个整数, 该ASCII串值表示一个十进制或十六进制的数。

语法

```
#include <stdlib.h>
unsigned int16 ahtoi16(char near * str);
unsigned int8 ahtoi8(char near * str);
```

```
int16 atoi16(char near * str);
int8 atoi8(char near * str);
```

描述

atoi16() 将一个以“空”字符结尾的ASCII串转换为一个int16整数值，该ASCII串值表示一个无符号的十六进制数。

atoi8() 将一个以“空”字符结尾的ASCII串转换为一个字的整数值，该ASCII串值表示一个无符号的十六进制数。

atoi16() 将一个以“空”字符结尾的ASCII串转换为一个带符号的int16整数值，该ASCII串值表示一个带符号的数。

字符串必须是下述形式之一。

-0b100000000000000000	到	0b1111111111111111	二进制
-0o100000	到	0o17777	八进制
-0100000	到	0177777	八进制
-32768	到	65535	十进制
-0x8000	到	0xffff	十六进制

atoi8() 将一个以“空”字符结尾的ASCII串转换为一个带符号的字的值，该ASCII串值表示一个带符号的数。

字符串必须是下述形式之一。

-0b10000000	到	0b11111111	二进制
-0o200	到	0o377	八进制
-0200	到	0377	八进制
-128	到	255	十进制
-0x80	到	0xFF	十六进制

6. qsort

名称

qsort() 对一个数组进行快速排序。

语法

```
#include <stdlib.h>
void qsort(void near * base, size_t nelem, size_t size);
```

描述

qsort() 对一个数组中的元素进行排序。这些元素留在原位置不动。

该函数接受一个指向数组的指针，以及数组中元素的个数(nelem)和每个元素的大小(size)。nelem和size是size_t类型，该类型在string.c中定义。

qsort() 使用一个外部函数比较数组中的元素，该外部函数必须按下述方式定义：

```
#define QSORT_COMPARE(arg1,arg2)
```

如果没有定义, QSORT_COMPARE将默认为string.c中的strcmp()。QSORT_COMPARE必须接受两个指针, 并且返回一个int8值。返回的值必须是下述三种情况之一:

- 如果第一个元素小于第二个, 则返回值小于0。
- 如果第一个元素等于第二个, 则返回值等于0。
- 如果第一个元素大于第二个, 则返回值大于0。

7. pow

名称

计算一个数的幂。

语法

```
#include <stdlib.h>  
unsigned int16 pow(unsigned int8 base, unsigned int8 exponent);
```

描述

该函数计算base (基数) 的exponent (指数) 次幂。



STRING

1. STRING.H和STRING.C

名称

对以“空”字符结尾的未知长度字符串执行操作。

描述

在此函数库中的例程对以“空”字符结尾的未知长度字符串缓冲器执行操作。

2. size_t

名称

size_t是变量的“大小”类型。

语法

```
#include <string.h>  
typedef unsigned int8 size_t;
```

描述

Byte Craft函数库接受“大小”参数作为size_t类型。size_t参数通常表示其他参数或目标的大小。

3. memcpy, memchr和memcmp

名称

memcpy(), memchr()和memcmp()拷贝、搜索和比较缓冲器。

语法

```
#include <string.h>
void memcpy(char near * dest,const char far * src,size_t n);
void * memchr(const void * s,int8 c,size_t n);
int8 memcmp(unsigned char far * str1,unsigned char far * str2,
            size_t n);
```

描述

memcpy()将n个内存字节从位置src拷贝到位置dest。

memchr()在一个数组中查找一个字符。它由位置s开始，在大小为n的数组中查找第一个等于c（无符号字符）的元素。它返回匹配元素的地址。如果没有找到相匹配的元素，则返回一个空指针。

memcmp()比较两个无符号字符组成的数组str1和str2，找出它们之间的不同。如果所有的元素是相等的，则memcmp()返回0。

两个数组有某处若不相同，则如果str1的元素大于str2的，则memcmp()返回一个正值。如果str1的元素小于str2的，则memcmp()返回一个负值。

两个数组的大小必须都是n。

4. strcat, strchr和strcmp

名称

strcat (), strchr ()和strcmp ()拷贝、搜索和比较以“空”字符结尾的字符串。

语法

```
#include <string.h>
void strcat(char near * dest,char far * src);
void * strchr(const void * str,int8 c);
int8 strcmp(unsigned char far * str1,unsigned char far * str2);
void strcpy(char near * dest,char far * src);
```

描述

strcat ()拷贝以“空”字符结尾的字符串src的元素，包括空字符，到数组dest中。

strchr()在以“空”字符结尾的字符串str中搜索第一个出现的c字符。strchr()将str的空字符结尾作为字符串的一部分进行检查。strchr()返回一个指向str中匹配字符的指针，如果没有找到相匹配的字符则返回一个空指针。

strcmp()比较两个以“空”字符结尾的串str1和str2，找出它们之间的不同。如果所有的元素是相等的，则strcmp()返回0。

两个字符串有某处若不相同，且如果str1的元素大于str2的，则strcmp()返回一个正值。如果str1的元素小于str2的，则strcmp()返回一个负值。

如果一个字符串比另一个短，则strcmp()不会对较长字符串的多出部分继续进行比较操作。

strcpy()拷贝以“空”字符结尾的字符串str，包括空字符，到字符数组dest中。

5. strlen

名称

strlen() 确定以“空”字符结尾的字符串的长度。

语法

```
#include <string.h>
unsigned int8 strlen(char far * str);
```

描述

strlen() 返回以“空”字符结尾的字符串str中字符的数目。总数不包括结尾的空字符。

6. strset, strupr和strlwr

名称

strset(), strupr() 和 strlwr () 重新初始化或转换以“空”字符结尾的字符串。

语法

```
#include <string.h>
void strset(char near * str, char ch);
void strupr(char near * str);
void strlwr(char near * str);
```

描述

strset存储unsigned char类型的字符ch到str指向数组的每一个元素。

strupr() 将以“空”字符结尾的字符串中的所有小写字母转换为大写字母。它在原位转换字符串。

strlwr() 将以“空”字符结尾的字符串中的所有小写字母转换为大写字母。它在原位转换字符串。

CTYPE

1. CTYPE.H

名称

ctype例程执行对字符的操作。

描述

在此函数库中的例程对字符执行类型鉴别和转换操作。

2. isxyz, toascii, tolower和toupper

名称

isalnum(), isalpha(), isascii(), iscntrl(), isdigit(), islower(), isupper(), isxdigit(), toascii(), tolower() 和 toupper() 检测和转换字符。

语法

```

#include <ctype.h>
int8 isalnum(int8 ch);
int8 isalpha(int8 ch);
int8 isascii(int8 ch);
int8 iscntrl(int8 ch);
int8 isdigit(int8 ch);
int8 islower(int8 ch);
int8 isupper(int8 ch);
int8 isxdigit(int8 ch);
#define toascii(CH) CH&0x7f
int8 tolower(int8 ch);
int8 toupper(int8 ch);

```



描述

`isalnum()` 检测字符 `ch`。如果 `ch` 是一个小写字母 (a~z)、大写字母 (A~Z) 或十进制数 (0~9)，则返回一个非零的值。如果不是上述字母或数字，则返回零。

`isalpha()` 检测字符 `ch`。如果 `ch` 是一个小写字母 (a~z)、大写字母 (A~Z)，则返回一个非零值。如果不是上述字母，则返回零。

`isascii()` 检测字符 `ch`。如果 `ch` 是一个 ASCII 字符 (高位为 0)，则返回一个非零值。

`iscntrl()` 检测字符 `ch`。如果 `ch` 是一个 ASCII 控制字符 (ASCII 控制字符包括字符 0~31 和 127)，则返回一个非零值。如果不是，则返回零。

`isdigit()` 检测字符 `ch`。如果 `ch` 是一个数字 (0~9)，则返回一个非零值。如果不是，则返回零。

`islower()` 检测字符 `ch`。如果 `ch` 是一个小写字母 (a~z)，则返回一个非零值。如果不是上述字母，则返回零。

`isupper()` 检测字符 `ch`。如果 `ch` 是一个大写字母 (A~Z)，则返回一个非零值。如果不是上述字母，则返回零。

`isxdigit()` 检测字符 `ch`。如果 `ch` 是一个十六进制数 (0~9, a~f 或 A~F)，则返回一个非零值。如果不是，则返回零。

`toascii()` 将 `CH` 的高位赋值为 0。

`tolower()` 检测字符 `ch`。如果 `ch` 是一个大写字母，则返回其相对的小写字母。如果是其他情况，则返回一个不变的 `ch`。

`toupper()` 检测字符 `ch`。如果 `ch` 是一个小写字母，则返回其相对的大写字母。如果是其他情况，则返回一个不变的 `ch`。

DELAY

1. DELAY.H 和 DELAY.C

名称

delay例程使嵌入式程序执行等待操作。

描述

此例程对延时请求提供统一的接口。

要求

部分头文件或定义文件。

2. delay_ms

名称

delay_ms()延迟几个毫秒。

语法

```
#include <delay.h>
void delay_ms(unsigned int8 ms);
```

描述

delay_ms()延迟指定数目的毫秒时间，然后返回。



KEYPAD

1. KEYPAD.H和KEYPAD.C

名称

keypad驱动一个矩阵键盘。

描述

该函数库中的例程对连接到单一的、8位I/O端口的矩阵键盘进行操作。

要求

端口和延时函数库。

2. keypad_getch和keypad_kbhit

名称

keypad_getch()和keypad_kbhit()扫描矩阵键盘和获取一个字符。

语法

```
#define KEYPAD_PORT
#define keypad_debounce_delay() delay_ms(0x20)
#include <keypad.h>
```

```
unsigned char keypad_getch(void);
```

```
unsigned int8 keypad_kbhit(void);
```

描述

用户必须定义KEYPAD_PORT给用于读入和写出到这一端口的寄存器。

可以是默认的定义，请参考键盘函数库的源代码。

用户必须定义一个KEYPAD_READ函数,以设置KEYPAD_PORT用于读入。具体实现将依赖于键盘的电路。

可以是默认的定义,依赖于您的CDS产品。请参考键盘函数库的源代码。

keypad_debounce_delay()由keypad_getch()调用。如果没有重新定义,Keypad_debounce_delay()等待20毫秒,这期间键盘无效。

keypad_getch()等待击键,并且返回相应的从数组keypad_table[]得到的字符。

如果没有在其他地方定义, keypad_table默认为标准电话键。

```
const char keypad_table[]="123A"
                        "456B"
                        "789C"
                        "*0#D";
```

keypad_kbhit()等待击键,并且在敲击后返回1。

LCD

1. LCD .H和LCD.C

名称

lcd提供对lcd控制器的支持。

要求

端口函数库。

描述

LCD函数库提供的例程能驱动Hitachi(日立公司)的HD44780 LCD控制器。

典型的LCD模块配置使用3条导线用于读/写、寄存器选择(命令或数据)和启用,4条或8条导线用于数据传输。

此模块需要由一系列导线初始化,这些导线用于设置参数,包括数据总线的宽度。这项工作由lcd_init()来完成。初始化之后,LCD显示屏偶而会处于忙状态。lcd_busy_check()决定此模块是否接受新的数据。

lcd_putchar()和lcd_getch()计划用作putch(),但getch()用于stdio函数库的可能性较小。

配置

lcd.h定义若干用于LCD软件命令的重要常量。

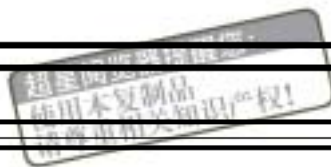
下述符号需要定义。在lcd.c中提供了默认值。

```
#define LCD_E_PORT PORT1    /* LCD 启用 */
#define LCD_E_PIN  2        /* LCD 启用 */
#define LCD_DATA    PORT1
#define LCD_RS_PORT PORT0    /* LCD 寄存器选择 */
#define LCD_RS_PIN  0        /* LCD 寄存器选择 */
```

```
#define LCD_RW_PORT PORT0    /* LCD 读/~写 */  
#define LCD_RW_PIN 1        /* LCD 读/~写 */
```

2. LCD_DATA

名称



```
void lcd_gotoXY(int8 x, int8 y);
```

描述

lcd_putchar() 写一个字符到LCD显示屏。

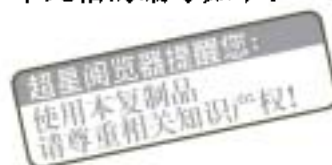
lcd_getch() 从LCD显示屏读一个字符。

lcd_gotoXY() 移动LCD插入点到特定字符单元格。单元格的编号如下：

```
X 0 1 2 3 4 5 6 7 8 9 ...
Y +-----+
  0|
  1|
  ...
```

要想移动插入点到2行、40列的显示屏的最底行的最后一个单元格，可使用语句：

```
lcd_gotoXY(1, 39)
```



I2C_EE

1. I2C_EE .H和I2C_EE.C

名称

I2C_EE为PC 24LC01B/02B串行EEPROM（电可擦除只读存储器）提供有用的例程。

描述

PC™是Philips Electronics N.V.的标准。它是一个通过两根导线操作的串行外围接口。这两条线是串行数据线和串行时钟线，二者都是双向的，并且是同步的。

它是带有冲突检测的多控制和多受控的网络接口。在此网络中存在的设备数目可达128个。每个设备都有一个由几个固定位（由PC委员会指定）和几个可编程位组成的地址，这几个可编程位由引脚连接确定。因此，几个相同设备可以并存于一个系统中。7位或10位地址都可用。

还有几个保留地址，用于向所有设备的广播和其他扩展性需要。

PC有两种速度：在标准模式下，100千比特/秒；在快速模式下，400千比特/秒。有效数据率取决于配置和使用的编址模式。

标准没有指定实现它的控制器的程序设计接口。这一部分专门处理PC连接的串行EEPROM。

要求

端口和延迟函数库。

配置

要配置PC端口，下述设置必须被调整。如果不改变的话，PC控制（时钟）线是端口1的位0，数据线是端口2的位5。

```
#define I2C_PORT_DDR_READ() GPIO_CONFIG = PORT0_RESISTIVE | \
PORT1_CMOS | PORT2_RESISTIVE | PORT3_RESISTIVE; PORT2=0xff
#define I2C_PORT_DDR_WRITE() GPIO_CONFIG = PORT0_RESISTIVE | \
```



```

PORT1_CMOS | PORT2_CMOS      | PORT3_RESISTIVE
#define I2C_PORT_DDR() GPIO_CONFIG = PORT0_RESISTIVE | \
PORT1_CMOS | PORT2_RESISTIVE | PORT3_RESISTIVE; PORT2=0xff

#define I2C_CONTROL PORT1
#define I2C_DATA PORT2
#define I2C_SCL 0
#define I2C_SDA 5

#define i2c_bus_delay() delay_ms(1)

```



2. I2C_write和I2C_read

名称

I2C_write()和I2C_read()在I²C总线上通信。

语法

```

#include <i2c_ee.h>
void I2C_write(unsigned int8 address, unsigned int8 data);
unsigned int8 I2C_read(unsigned int8 address);

```

描述

I2C_write()写一个字data到位于串行EEPROM的内存单元address上。

I2C_read()读取内存单元address上的值。

MWIRE_EE

1. MWIRE_EE.H和MWIRE_EE.C

名称

mwire_ee创建一个到串行EEPROM上的MICROWIRE连接。

描述

MICROWIRE和MICROWIRE/PLUS是(美)国家半导体(National Semiconductor)协会的专有标准。在一些实现中,它与SPI是兼容的。

MICROWIRE/PLUS是一个通过三条导线操作的串行外围接口。它是同步的,并依赖于内部(到总线主控)或外部时钟。它是双向的。芯片选择信号必须被实现。

有下述程序设计接口。

- 配置接口(包括内部产生的移位速率)的控制寄存器CNTRL。
- 读/写串行输入/输出寄存器SIOR。

这些寄存器是内存映射的。

MICROWIRE移位时钟(SK)是内部时钟速度的基本单位,可用2、4或8进一步分割系统时钟。MICROWIRE传输和接收一个字节需要8个SK周期。

通过设定PSW（处理器状态字）的BUSY标志位，软件可以引发一次传输。当传输完成后，BUSY标志位将清零。当BUSY复位时，一些部件提供一个向量可屏蔽的中断。

下述例程直接处理通过MICROWIRE连接的EEPROM。

要求

设备头文件或定义文件，以及在下面列出的外部函数。

配置

在使用mwire_ee函数库之前，必须定义下述符号。如果没有定义，将使用默认值。

MWIRE_CONTROL	用于访问MICROWIRE控制线的端口
MWIRE_CLK	用于时钟的引脚
MWIRE_CK	用于芯片选择的引脚
MWIRE_DATA	用于访问MICROWIRE数据线的端口
MWIRE_DO	用于数据输出的引脚
MWIRE_DI	用于数据输入的引脚
MWIRE_PORT_DDR_READ()	用于读的设置端口数据方向的宏
MWIRE_PORT_DDR_WRITE()	用于写的设置端口数据方向的宏
MWIRE_PORT_DDR()	用于MICROWIRE端口的设置默认数据方向的宏

2. mwire_bus_delay

名称

mwire_bus_delay()是用户定义的延时函数。

语法

```
#include <mwire_ee.h>
void mwire_bus_delay() {
/* 您喜欢的延时代码 */
}
```

描述

为了适当地为MICROWIRE总线安排时间，必须写一个在半个时钟周期之间等待的延时函数。可以通过下述方式完成此项工作。

- 将其定义成包含NOP的函数。
- 将其定义成调用延时函数的函数。

3. mwire_enable, mwire_disable, mwire_write, mwire_read和mwire_write_all

名称

mwire_enable(), mwire_disable(), mwire_write(), mwire_read()和mwire_write_all()通过MICROWIRE通信。

语法

```
#include <mwire_ee.h>
```

```
#define mwire_enable()  
#define mwire_disable()  
#define mwire_erase(ADDRESS)  
void mwire_write(unsigned int8 address,unsigned int16 data);  
unsigned int16 mwire_read(unsigned int8 address);  
void mwire_write_all(unsigned int16 data);
```

描述

mwire_enable()和mwire_disable()分别启用和禁止连接到串行EEPROM的MICROWIRE。

mwire_erase()擦除串行EEPROM上的内存单元ADDRESS上的值。

mwire_write()在串行EEPROM上的内存单元address上写入值。

mwire_read()读出并返回串行EEPROM上内存单元address上的值。

mwire_write_all()在串行EEPROM上的所有单元上写入同一个值。

MATH

1. MATH.H和MATH.C

名称

math实现数学函数。

描述

该函数库实现数学函数。

要求

float.h。

2. acos, asin, atan和atan2

名称

acos(), asin(), atan()和atan2()是三角函数。

语法

```
#include <math.h>  
float accs(float x);  
float asin(float x);  
float atan(float x);  
float atan2(float y, float x);
```

描述

acos()返回一个弧度表示的角度值(从0 ~ π)，该角度的余弦值是x。

asin()返回一个弧度表示的角度值(从 $-\pi/2$ ~ $\pi/2$)，该角度的正弦值是x。

atan()返回一个弧度表示的角度值(从 $-\pi/2$ ~ $\pi/2$)，该角度的正切值是x。

`atan2()` 返回一个弧度表示的角度值 (从 $-\pi \sim \pi$)，该角度的正切值是 y/x 。

3. `ceil` 和 `floor`

名称

`ceil()` 和 `floor()` 返回下一个较高或较低的整数值。

语法

```
#include <math.h>
float ceil(float x);
float floor(float x);
```

描述

`ceil()` 返回 x (如果是一个整数)，或者下一个较高的整数值。

`floor()` 返回 x (如果是一个整数)，或者下一个较低的整数值。

4. `cos` 和 `cosh`

名称

`cos()`，`cosh()`，`sin()`，`sinh()`，`tan()` 和 `tanh()` 是三角函数。

语法

```
#include <math.h>
float cos(float x);
float cosh(float x);
float sin(float x);
float sinh(float x);
float tan(float x);
float tanh(float x);
```

描述

`cos()` 返回 x 的余弦值， x 是一个弧度表示的角度值。

`cosh()` 返回 x 的双曲余弦值。

`sin()` 返回 x 的正弦值， x 是一个弧度表示的角度值。

`sinh()` 返回 x 的双曲正弦值。

`tan()` 返回 x 的正切值， x 是一个弧度表示的角度值。

`tanh()` 返回 x 的双曲正切值。

5. `fabs`

名称

`fabs()` 计算一个浮点数的绝对值。

语法

```
#include <math.h>
float fabs(float x);
```



描述

`fabs()` 返回x的绝对值。

6. fmod**名称**

`fmod()` 计算x/y的余数。

语法

```
#include <math.h>
float fmod(float x, float y);
float frexp(float x, int * pexp);
float ldexp(float x, int exp);
```

**描述**

`fmod()` 返回x/y的余数。

`frexp()` 计算浮点数x的尾数和指数。`frexp()` 返回尾数，并且将指数放在变量*`pexp`中。这个指数是2的乘幂。

`ldexp()` 计算尾数x及指数exp（以2为底数）对应的浮点数值。

7. exp, log和log10**名称**

`exp()`, `log()`和`log10()` 计算指数和对数。

语法

```
#include <math.h>
float exp(float x);
float log(float x);
float log10(float x);
```

描述

`exp()` 返回x的指数次幂（即e增加到x次幂）。

`log()` 返回x的自然对数。

`log10()` 返回x的以10为底的对数。

8. modf**名称**

`modf()` 计算浮点数的整数和小数部分。

语法

```
#include <math.h>
float modf(float x, float * pint);
```

描述

modf() 计算数值x的整数和小数部分，返回小数部分，并将整数部分存储在变量*pint中。整数和小数部分的正负号与x相同。

9. pow和sqrt**名称**

pow() 和 sqrt() 计算浮点数的乘幂或平方根。

语法

```
#include <math.h>
float pow(float x, float y);
float sqrt(float x);
```

描述

pow() 返回y次幂后的x。

sqrt() 返回x的平方根。

超星浏览器提醒您：
使用本复制品
请尊重相关知识产权！

FLOAT**FLOAT.H****名称**

float是浮点数定义的函数库。

语法

```
#include <float.h>
#define FLT_DIG
#define FLT_EPSILON
#define FLT_MANT_DIG
#define FLT_MAX
#define FLT_MAX_10_EXP
#define FLT_MAX_EXP
#define FLT_MIN
#define FLT_MIN_10_EXP
#define FLT_MIN_EXP
#define FLT_RADIX
#define FLT_ROUNDS
```

描述

如果使用浮点变量或操作，文件float.h提供一些要求的定义。

定义

FLT_DIG() 确定浮点变量精度的位数。

FLT_EPSILON 确定浮点变量最小的可能的非零值。

FLT_MANT_DIG()是浮点变量尾数的位数。值是基数FLT_RADIX。

FLT_MAX确定浮点变量的最大的可能值。

FLT_MAX_10_EXP是一个整数指数。当FLT_RADIX增加到FLT_MAX_10_EXP次幂时，其结果是浮点变量的最大的10次幂值。

FLT_MAX_EXP是一个整数指数。当FLT_RADIX增加到FLT_MAX_EXP-1次幂时，其结果是浮点变量的最大的FLT_RADIX次幂值。

FLT_MIN提供浮点变量的最小的可能值。

FLT_MIN_10_EXP是一个整数指数。当FLT_RADIX增加到FLT_MIN_10_EXP次幂时，其结果是浮点变量的最小的10次幂值。

FLT_MIN_EXP是一个整数指数。当FLT_RADIX增加到FLT_MIN_EXP-1次幂时，其结果是浮点变量的最小的FLT_RADIX次幂值。

float类型变量的指数是一个FLT_RADIX指数。

FLT_ROUNDS表示用于浮点数计算的取整方法。下述FLT_ROUNDS值设置了伴随的取整方法：

- 1 编译器将取整为一个最接近的可表示数值。

UART

1. UART

名称

UART在软件中提供UART（通用异步收发器）函数。

要求

部分头文件或定义文件，以及端口和延时函数库。

定义

UART操作要求下述设置。

UART_TD_PORT

用户将它定义为用于UART传输的端口。默认时定义为PORT1。

UART_TD_PIN

用户将它定义为用于驱动TD线的UART_TD_PORT中的引脚。默认时定义为1。

UART_RD_PORT

用户将它定义为用于UART接收的端口。默认时定义为PORT2。

UART_RD_PIN

用户将它定义为用于读取RD线的UART_RD_PORT中的引脚。默认时定义为4。

变量

uart_mode

在运行时间配置uart函数库，具体如下所述。

设置

用户必须用OR联接的常数来设置uart_mode变量。

波特率	停止位	奇偶位	数据位
BAUD_300	STOP_1	PARITY_NONE	DATA_7
BAUD_1200	STOP_2	PARITY_EVEN	DATA_8
BAUD_2400		PARITY_ODD	
BAUD_4800			
BAUD_9600			
BAUD_19200			
BAUD_38400			
BAUD_57600			
BAUD_115200			



例子

```
uart_mode=BAUD_115200 | STOP_2 | PARITY_NONE | DATA_8;
```

2. uart_getch, uart_putch和uart_kbhit

名称

uart_getch(), uart_putch()和uart_kbhit()执行UART I/O。

语法

```
char uart_getch(void);
void uart_putch(char);
char uart_kbhit(void);
```

描述

uart_getch()从UART获取一个字符。

uart_putch()输出一个字符到UART。

uart_kbhit()如果接收到一个字节，则返回1。如果没有接收到数据，则返回0。

PORT

1. PORT.H, PORT.C和PORTDEFS.H

名称

port提供与平台无关的端口访问。

要求

部分头文件或定义文件。

描述

该头文件包括一些操作端口的有用函数。多数Byte Craft 的函数库依赖于这些定义。

所有单芯片MCU有某一特性的I/O端口。该函数库设法消除这些特性之间的差别。

port.h引发读入portdefs.h。portdefs包括对数据方向寄存器每一个可能设置的定义。在这些定义中，‘I’代表“输入”，‘O’代表“输出”。这就解决了何种状态（0或1）代表输入或输出的问题。例如：

```
/* DDR 使用 1 代表输出，0代表输入 */
#define 00000000 0b11111111
#define 00000001 0b11111110
/* ... 等等 ... */
#define 00001111 0b11110000
/* ... 等等 ... */
#define 11111110 0b00000001
#define 11111111 0b00000000
```



portdefs还包括用于DDR_MASKED()的位屏蔽定义。在这些定义中，‘_’（下划线）代表“没有改变”，而‘c’代表改变。

2. DDR(), DDR_MASKED()和DDR_WAIT()

名称

DDR(), DDR_MASKED()和DDR_WAIT()操纵端口的数据方向。

语法

```
#include <port.h>
DDR(port, direction)
DDR_MASKED(port, mask, direction)
DDR_WAIT()
```

描述

这些函数操纵端口的数据方向。它们使用从portdefs.h读入的方向和任务定义。

DDR()接受端口和方向定义，并且设置此端口的数据方向寄存器按此操作。

DDR_MASKED()执行相同的动作，但仅对于在掩码定义中选择的针脚执行操作。

DDR_MASKED()帮助解决一些函数库例程之间的冲突，这些函数库例程在相同端口上的不同位上寻址。为了改变一个或两个位，编译器可能使用有效的位改变指令，其他不涉及的位不变。另外，在编译器读入和修改DDR值的时候，它将保留外部掩码DDR位的状态。

DDR_WAIT()插入短暂的延时，以允许数据传播方向改变。

例子

如果要设置端口PORTX的位全部为输出，则调用下述语句：

```
DDR(PORTX, 00000000); /* 注意是字母“O”，而不是数字“0” */
DDR_WAIT();
```

如果要设置低4位和高4位分别为输出和输入，则使用下述语句：

```
DDR(PORTX, 11110000); /* 注意是字母“I”和“O” */
```

```
DDR_WAIT( );
```

如果要设置PORTX仅一个比特为输出，则使用下述语句：

```
DDR_MASKED(PORTX, _____C_, 00000000); /* 其他字母“0”位无关紧要*/  
DDR_WAIT( );
```



附录B ASCII 码 表

当想有一个ASCII码表时，总是很难找到。这里给出一个十六进制值和其对应ASCII码的ASCII码表。

表B-1 ASCII字符

HEX	ASCII	HEX	ASCII	HEX	ASCII	HEX	ASCII
00	NUL	20	SP	40	@	60	`
01	SOH	21	!	41	A	61	a
02	STX	22	"	42	B	62	b
03	ETX	23	#	43	C	63	c
04	EOT	24	\$	44	D	64	d
05	ENQ	25	%	45	E	65	e
06	ACK	26	&	46	F	66	f
07	BEL	27	'	47	G	67	g
08	BS	28	(48	H	68	h
09	HT	29)	49	I	69	i
0A	LF	2A	*	4A	J	6A	j
0B	VT	2B	+	4B	K	6B	k
0C	FF	2C	,	4C	L	6C	l
0D	CR	2D	-	4D	M	6D	m
0E	SO	2E	.	4E	N	6E	n
0F	SI	2F	/	4F	O	6F	o
10	DLE	30	0	50	P	70	p
11	DC1	31	1	51	Q	71	q
12	DC2	32	2	52	R	72	r
13	DC3	33	3	53	S	73	s
14	DC4	34	4	54	T	74	t
15	NAK	35	5	55	U	75	u
16	SYN	36	6	56	V	76	v
17	ETB	37	7	57	W	77	w
18	CAN	38	8	58	X	78	x
19	EM	39	9	59	Y	79	y
1A	SUB	3A	:	5A	Z	7A	z
1B	ESC	3B	;	5B	[7B	{
1C	FS	3C	<	5C	\	7C	
1D	GS	3D	=	5D]	7D	}
1E	RS	3E	>	5E	^	7E	~
1F	US	3F	?	5F	_	7F	DEL



附录C 术 语 表

累加器 (accumulator) 还有“A”、“AC”或其他名称。这个寄存器保留ALU（算术与逻辑部件）的操作结果。

模/数 (A/D) 模数转换。

寻址方式 (addressing mode) 用于确定在CPU中的内存单元，以及表示它的符号。

算术与逻辑部件 (ALU) 执行基本数学计算。例如，加、减、补码、否、AND（与）和OR（或）。

AND 逻辑运算符，如果两个操作数都是1则AND的结果为1。

ANSI C 美国国家标准协会的C语言标准。

汇编语言 (assembly language) 一种特殊机器语言的助记形式。

存储单元 (bank) 由寻址方式和其约束所确定的一个内存逻辑单元。

比特域 (bit field) 作为一个单元考虑的一组位。如果编译器支持，比特域可能越过字节的界限。

块 (block) 由大括号{ }围住的任何一部分C代码。块在方法上等同于单条指令，但加入了新的变量范围。

断点 (breakpoint) 为停止执行程序代码而设定的一个位置。断点用于调试程序。

控制器区域网络 (CAN) 控制器区域网络是由Bosch和Intel发展起来的。它是一个链接到被控制设备的内部模块总线。

类型转换 (cast) 也称为强制。即将一个变量从一种类型转换为另一种类型。

校验和 (checksum) 将指定的二进制值相加的结果。校验和常用于验证一个二进制数序列的完整性。

合适的计算机操作 (computer operating properly) 也称为COP。在可疑的执行状态下复位微控制器功能的外围设备或函数。COP作为一个词是来源于国家半导体协会的COP8微控制器产品系列的名称。

交叉汇编程序 (cross assembler) 在一种类型计算机上执行，而为另一种不同目标计算机汇编源代码的汇编程序。例如，运行在Intel x86上，而生成用于Motorola的68HCO5的目标代码的汇编程序。

交叉编译程序 (cross compiler) 在某种类型计算机上运行，而为另一种不同目标计算机编译源代码的编译程序。例如，运行在Intel x86上，而生成用于Motorola的68HCO5的目标代码的编译程序。

调试程序 (debugger) 一种帮助进行系统调试工作的程序，找出程序哪里有错误并修正。调试程序支持中断、转储和内存修改等特性。

声明 (declaration) 类型、名称和变量可能取值的说明。

间接访问 (dereference) 也记为 * 或 indirection。即获取一个指针指向的值。

EEPROM 电可擦除可编程只读内存。

嵌入式 (embedded) 固化在周围的系统和单元中。即在一个特殊环境中实现一个特殊的功能。

字节次序 (endianness) 多字节数据存储规范之间的区别。小端派将最低位首先存储在内存中。大端派将最高有效位首先存储在内存中。

全局变量 (global variable) 可以被程序任何一部分读取和修改的变量。

滞后现象 (hysteresis) 控制开关动作和其效果之间的时延。可以被加强, 以预防在控制状态下的快速短期反转。

变址寄存器 (index register) 也称为“X”或其他名称。此寄存器用于保存一个值, 该值是在变址寻址模式下的一个因子。经常用于数学运算, 尽管没有累加器那么多的功能。

中断 (interrupt) 发送到CPU要求服务的信号。本质上是正常执行流程之外的一个子程序, 但有许多额外的考虑。

J1850 由SAE (美国汽车工程师学会) 签署的一个内部模块。

局部变量 (local variable) 仅仅被程序某一特定模块或某些模块使用的变量。

逻辑操作符 (logical operator) 对其操作数执行逻辑操作的操作符。例如, !、&& 和 ||。

机器语言 (machine language) 能被特殊CPU理解的二进制代码指令。

掩码 (mask) 当使用逻辑操作符时, 为设置和清除在其他位组中的特殊位置而设计的一组位。

可屏蔽中断 (maskable interrupt) 软件可以激活和去活的中断。

内存映射 (memory-mapped) 与内存中的实际地址相关联的一个虚拟地址或设备。

无操作指令 (NOP) 无操作。用于建立延迟的指令。

非 (NOT) 逻辑否操作, 将0变成1, 而将1变成0。

目标代码 (object code) 表示成二进制数的非可执行形式的机器语言的指令。多个目标文件连接在一起形成可执行文件。

操作符 (operator) 表示对操作数实行操作的一个符号。例如, +、* 和 /。

或 (OR) 一种布尔操作, 当任一个操作数为1时其操作结果为1。

内存分页 (paging) 一页指一块逻辑内存。分页内存系统使用页地址, 位移地址指的是一个特殊内存单元。

端口 (port) 物理的输入/输出连接。

程序计数器 (program counter) 也称为PC。保存下一步将被执行的指令的寄存器。当每条指令的每一个字节被取出后, 程序计数器加1。

程序员模型 (programmer's model) 对多个寄存器的描述, 它们组成微处理器的可视界面。这些寄存器包括: 累加器、变址寄存器、程序计数器和栈指针。

可编程只读内存 (PROM) 即能被编程的ROM。

实时 (real time) 能以与实际事件发生时间相当的速度响应的系统。

寄存器 (register) 存在于专用的微处理器中的一个内存字节或字。与外部RAM相反, 寄

寄存器直接接到ALU（算术与逻辑部件）和其他微处理器功能。

复位（reset） 将微控制器返回到已知状态。该项操作会或许不会改变处理器的寄存器、内存和外围设备的状态。

ROM 只读内存。

能用于ROM的（ROMable） 如果置入ROM中，则将被执行的代码。

RS-232 标准串行通信端口。

SCI 也称为UART（通用异步收发机）。SCI是一个异步串行接口。信号的速度控制与RS-232串行标准兼容，但这一电气标准仅仅是电路板级的。

串行外围设备接口总线（SPI） 电路板级串行外围设备总线。

范围（scope） 变量的范围指程序的区域，这此区域中可访问该变量。

移位（shift） 也称为“旋转”，但两者之间有微小的差别。移位向左或向右按位移动寄存器的内容。

副作用（side-effect） 对一个变量无意的改变，或者指与函数返回值的计算没有直接关系的函数中指令的工作。

模拟器（simulator） 重新构建与一个硬件设备有相同输入和输出行为的程序。

栈（stack） RAM的一部分，用于存储临时数据。栈具有“后入先出”（LIFO）的结构。

栈指针（stack pointer） 包含有栈顶端地址的寄存器。

静态（static） 存储于RAM保留区域，而不是在栈中的变量。这一保留区域不能被其他变量使用。

定时器（timer） 独立于程序执行的外围计数设备。

通用异步收发机（UART） 串行转为并行，或并行转为串行的转换器。

易失性（volatile） 指一个数值出乎意料地改变的性质。随着时间的过去，编译器不能相信一个易失变量的值仍保持不变，因此不能进行某种优化。解决的办法是，由程序员明确地声明，或由编译器决定。

监视定时器（watchdog(timer)） 计算机正常操作电路的另一个名称。